Evolution on FPGAs for Feature Extraction

Reid Porter, B.E. (Hons), B.Inf.Tech. (Hons)

Submitted for the degree of Ph.D.
2001

Queensland University of Technology
School of Electrical and Electronic Systems Engineering
Cooperative Research Centre for Satellite Systems

Keywords

Cellular Automata, Custom Computers, Evolutionary Algorithm, Evolvable Hardware, Feature Extraction, FPGA, Genetic Algorithm, Image Processing, Morphological Networks, Multi-Spectral, Neural Networks, Pattern Recognition, Reconfigurable Computers, Remote Sensing, Stack Filters.

Abstract

Evolvable hardware circuits are presented for solving pattern recognition problems in image data sets, especially for feature extraction in remotely sensed multi-spectral imagery. The circuits are targeted at Field Programmable Gate Arrays (FPGAs). FPGAs are digital circuits that can be configured to meet application specific computational needs by downloading a configuration bit-string. Evolutionary Algorithms are optimization techniques that are unique in their simplicity and therefore their flexibility. This flexibility has led to a hardware design methodology known as Evolvable Hardware, which uses Evolutionary Algorithms to explore novel hardware solutions. FPGAs are ideal targets for Evolvable Hardware because the programming bit-string is a good match to the genetic bit-string used in Evolutionary Algorithms. However, experience has shown that evolving hardware solutions using the raw configuration bit-string is often impractical due to the large design space that must be explored.

Evolvable FPGA circuits are described where the design space is severely constrained to an interconnected array of meaningful high-level operators. First, novel variants of Cellular Automata are evolved on FPGAs to solve binary pattern recognition problems. These experiments constrain the FPGA bit-string to a meaningful network of Cellular Automata building blocks. This means problem specific constraints can be more easily implemented, which leads to a smaller, more relevant design spaces. The search space is then further constrained to enable the Cellular Automata architecture to be applied to gray-value image processing, This leads to a class of non-linear filter known as Stack Filters. Stack filters are found to have several properties desirable to FPGA implementation, but lack sufficient computational power to solve practical gray value texture classification problems.

Second, evolvable network architectures are implemented on FPGAs to solve practical feature extraction problems that are found in multi-spectral images. These experiments constrain the FPGA bit-string to a pipelined array of high-level nodes. The computational power of the Stack Filter is enhanced by considering the set of

Generalized Stack Filters, which leads to the class of Morphological Networks. A Morphological Network node is approximately one quarter the size of a Neural Networks node when implemented on FPGAs. However, the Neural Network has superior performance on multi-spectral feature extraction problems in experiment. Both networks perform poorly on broad area features that include many spectral signatures.

Third, a novel network node is proposed that addresses this problem by exploiting both spectral and spatial information. The node includes both Morphological and Neural Network functionality. By using high-level network building blocks, the design space is directed towards solutions that are particularly useful for the feature extraction problem. This includes a rich design space of linear and non-linear filters from traditional image processing algorithms.

Finally, the node is used to build multi-layered networks and is applied to a variety of multi-spectral feature extraction problems. An advanced evolutionary algorithm is used to optimize the network. Once trained, the network can be applied to large image data sets with over two orders of magnitude speed-up compared to software implementations. Promising results are found in comparing the evolvable network architecture with advanced spatio-spectral software solutions and more traditional techniques. Results also indicate there is room for future research, and the directions considered most fruitful are described.

TABLE OF CONTENTS

List of Figures

List of Publications

Chapter 3

Porter, R. and N. Bergmann, *A generic implementation framework for FPGA based stereo matching*. In TENCON'97: IEEE Region 10 Annual Conference. Speech and Image Technologies for Computing and Telecommunications. 1997. Brisbane, Australia.

Porter, R., *Implementing GA Accelerators using FPGAs*. In GECCO-99: Student workshop of the Genetic and Evolutionary Computation Conference. July 1999. Orlando, Florida.

Porter, R, K. McCabe and N. Bergmann, *An Applications Approach to Evolvable Hardware*. In 1st Nasa/DoD Workshop on Evolvable Hardware. July 1999. Pasadena, California.

Chapter 4

Porter, R. and N. Bergmann, *Evolving FPGA Based Cellular Automata*. In SEAL'98: Simulated Evolution and Learning. October 1998. Canberra, Australia. Springer-Verlag.

Chapters 5 and 6

Porter, R. M. Gokhale, N. Harvey, S. Perkins and C. Young, *Evolving Network Architectures using Custom Computers for Multi-Spectral Feature Identification.* Accepted for publication in 3rd Nasa/DoD Wokshop on Evolvable Hardware. July 2001. Long Beach, California.

Appendix B

Perkins, S., R. Porter and N. Harvey, *Everything on the Chip: A Hardware-Based Self-Contained Spatially-Structured Genetic Algorithm for Signal Processing*. In Evolvable Systems: From Biology to Hardware. 2000. Scotland, UK: Springer Verlag.

Declaration

Reid Porter
May 2001

Acknowledgements

This thesis is based on 4 years of research on my part, but would not have been possible without inspiration, education and financial support from a large number of people.

My first two years of research at Queensland University of Technology was made possible by generous support from the Australian government and particularly the Co-operative Research Center for Satellite Systems. Thanks to my supervisor Neil Bergmann whose ability to see the point, even when provided with nothing but conceptual hunches and awkward associations, never ceases to amaze me! Thanks also to Mohammed Bennamoun for cherry smiles and good advice.

The last two years of this thesis brought me to Los Alamos National Laboratory, and particularly the Space and Remote Sensing Sciences group. It has been an amazing opportunity, and my brain continues to reel with the stimulus it receives from so many people. Thanks John Szymanski for thinking of my future and providing remote sensing insights. Thanks Jeff Bloch for encouraging new ideas by setting example through....as you say, a manager who codes! Thanks to Maya Gokhale, who has given me significant freedom, been nothing but encouraging and provided clear direction many times.

Thanks to the GENIE team, there's the source: a fountain of information at my disposal. Neal Harvey of course for morphology, a local library, and the Semi Autonomous Machine (SAM) to observe. Simon Perkins, whose clarity of explanation conveyed more information to me in these last two years than I could possibly have learnt by myself. Cody Young for crazy conversations and ideas that will prove the long term winners! James Theiller for keeping me honest, and proof that creative ideas and well-grounded science are not mutually exclusive! Steven Brumby for drawing connections and great P.R.(PAC-8 goes national?).

Thanks Chisty of course who not only contributed significantly in thoughts and ideas, but also kept me loved and happy and gives the whole thing point.

Chapter 1

Introduction

This thesis, at heart is about building hardware for gray value, color and multi-spectral image processing. This is not a new thing, and many researchers throughout the world continue to work on this problem from a great many directions. Words by Kendall Preston are particularly appropriate to the underlying approach of this thesis:

“Digital filters for image analysis are often considerably different from those in other areas… Speed, simplicity, and low cost are the primary requisites.” (1983) [1].

The experiments of Preston provide a clear demonstration of solving problems with the resources at hand; that is, trying to find solutions to problems that can be implemented with the smallest number of building blocks available in digital hardware.

This thesis focuses on the problem of Automatic Feature Extraction[1] (AFE) in image data sets. AFE attempts to find algorithms that can consistently separate a feature of interest from the background in the presence of noise and uncertain conditions. In a more general sense, this is an optimization problem. Preston's minimalist approach is particularly attractive within the context of optimization problems. It may be difficult to apply a fixed set of hardware building blocks to a particular problem because algorithms are usually very abstracted from the hardware. It is possible however that an optimal arrangement of the fixed set of building blocks may be sufficient for solving the problem.

[1] While Pattern Recognition may be considered a more technically appropriate term, AFE is used throughout this thesis since it is indicative of the type of pattern recognition problem addressed i.e. finding features of interest in multi-spectral data sets.

In this thesis the building blocks are Field Programmable Gate Arrays (FPGAs). A number of FPGAs are combined, usually with local memory, on a plug-in board that is called a Custom Computer (CC). Software applications exploit CC by designing custom circuits for the computationally intensive parts of an algorithm. The host and CC are usually tightly coupled through a global bus, and therefore, an application can benefit from the flexibility of software as well as the performance of the specialized hardware.

This flexibility of CC has been identified as particularly useful in optimization problems [2]. This is because the implementation requirements of optimization problems usually vary from one problem to the next. The optimization techniques used in this thesis, and for which this is particularly true, are Evolutionary Algorithms (EA). EA are computationally intensive but are unique in their flexibility. EA can be used to optimize almost anything or any structure for a particular problem. This flexibility has led to their wide spread use in the field of evolvable hardware. Figure 1 illustrates how EA design is often considered in the field of Evolvable Hardware.

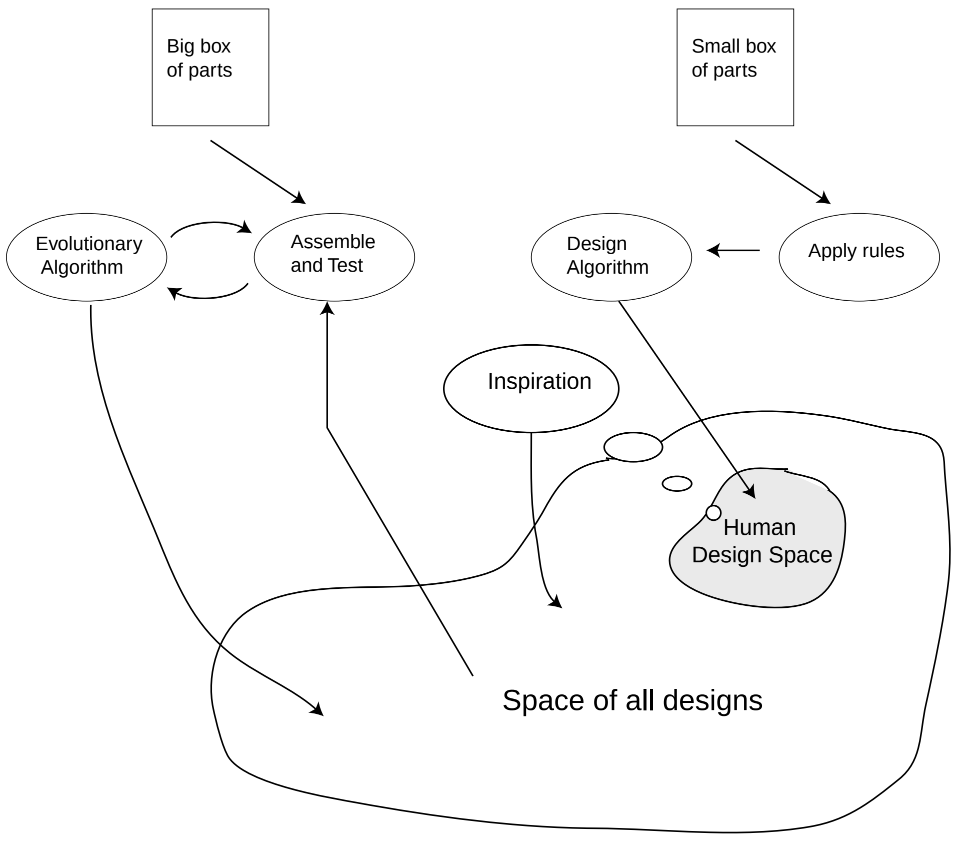


Figure 1: EA compared to conventional assemble-and-test design [3].

Traditional *assemble-and-test* of digital circuits applies top-down rule based approaches. Often assumptions are made, and constraints imposed in order to make solutions tractable. These constraints lead to solutions from a sub-space of what can be considered the total space of all possible solutions. In the EA *assemble-and-test,* the search can be performed without constraints and the larger space explored. In this thesis, EA are used to explore this larger space of designs for *hardware efficient* solutions to AFE problems. Before this approach can be taken, it is important to realize that constraints introduced by human designers are not necessarily a bad thing. In fact, it can be argued that many human design spaces, particularly those found through human inspiration, are particularly useful for solving a problem. It is therefore important to be familiar with traditional design spaces. Once these are understood, constraints can then be relaxed and hardware efficient solutions found.

A problem of particular interest to this thesis is AFE in multi-spectral, remotely sensed imagery. Increasing numbers of earth observing satellites are collecting raw image data. At the same time, sensor technology is rapidly developing. Images are now taken at ten to hundreds of spectral channels and at increasing spatial resolutions. The typical *image cube*[2] has an order of magnitude more data than single channel gray-scale or 3 channel color data sets, and is likely to grow. To find features of interest within these large amounts of data is a time consuming and costly process and therefore AFE is an attractive alternative.

This thesis develops high-throughput AFE algorithms in hardware. Several advantages of implementing an AFE algorithm in hardware are immediately obvious:

- Data from high-volume satellite sensors is usually collected in large image databases in a ground-based station. Current AFE algorithms, based in software, pose a computational bottleneck in capitalizing on this large data resource. Hardware AFE would enable these large databases to be searched and cross-referenced at speeds an order of magnitude greater than software implementations.
- Data flags specifying regions of interest could be automatically generated to reduce the amount of raw imagery a human analyst has to inspect.

[2] An image with multi-spectral channels can be visualized as an image cube by stacking the images of different wavelengths on top of each other.

- Content-based indices for user- defined features of interest could be generated on demand at useful speeds.
- Eventually AFE hardware could be moved to the sensor and high level information extracted as the data is collected.
- Real-time extraction of high-level information would also be benefit to secondary sensors as a means to narrow search parameters and enable real-time response in a dynamic environment.
- As satellite data volumes increase, so does the bandwidth requirements for communicating data to the ground station. Eventually high-level information from AFE could be used to choose the most relevant information for selective download across a limited communications channel.

In order to capitalize on these advantages, there are several problems that must be addressed. Traditionally, AFE algorithms are developed in software, and then if possible, ported to dedicated hardware systems. This approach suffers from long development times and is therefore rarely seen in practice. This thesis investigates an alterative, suggested by the evolvable hardware paradigm, and attempts to answer the question: can we find hardware solutions to AFE problems directly? In answering this question, this thesis will investigate which design spaces are most suitable for *hardware efficient* AFE. Problems faced in traditional software AFE are also considered in this thesis, although to a lesser degree than a PhD focused on software solutions. These problems include: Does the AFE hardware have sufficient complexity to fit the training data? How does optimized hardware perform on out-of-training data, or how well does it generalize? Generalization is perhaps the most important aspect of AFE problems and is critical to the eventual goal of automatic intelligent processing of large volume data sets. While this thesis does not attempt to address theoretical aspects of generalization (a topic that has made significant advances in recent years), quantitative testing of generalization is used throughout the thesis to guide design choices.

The rest of the thesis

The rest of the thesis is now described chapter by chapter. This is to direct readers towards particular aspects that they may find most interesting. Introductions at the start of each chapter and Chapter Summaries at the end of each chapter provide a more detailed roadmap, which readers may find useful.

Chapter 2, Literature Review: Three fields of study are described. Section 2.1 defines the problem: Automatic Feature Extraction and describes the conventional approaches, or human inspired design spaces that are currently used. Section 2.2 provides an introduction to Evolutionary Algorithms. Their application to AFE is described in Section 2.2.4. In Section 2.3 the hardware building blocks are described both in terms of Field Programmable Gate Arrays, and Custom Computer architectures. Section 2.3.3 describes work in evolvable hardware and other research efforts where EA has been used in combination with CC. The Chapter Summary in Section 2.4 provides a more detailed description of the thesis approach with respect to reported literature.

Chapter 3, Design Considerations: This chapter is of most interest to a reader who wishes to implement EA experiments using custom computers. It provides introduction to the major design choices as well as more detailed description of architectures used in this thesis.

Chapter 4, Evolving Cellular Architectures: This chapter is dictated largely by the architecture of the XC6216 FPGA. This FPGA does not have sufficient resources to solve practical AFE problems, and therefore experiments explore more theoretical architectures. These architectures include cellular automata in Section 4.1 and stack filters in Section 4.2. Experimental conclusions, which are presented in the Chapter Summary, provide a firm foundation for the approach used in subsequent chapters.

Chapter 5, Evolving Network Architectures: In this chapter, the development of practical AFE hardware begins. It introduces network architectures and explains why they are desirable in hardware implementations. This chapter implements and compares two types of network architecture. The first is Neural Networks, which have received considerable attention for solving problems related to AFE. The second is Morphological Networks, a recently introduced variant that have a close relationship to many algorithms in image processing, and can be more efficiently implemented on FPGAs than Neural Networks.

Chapter 6, Evolving Multi-Spectral Networks: This chapter represents the major design contribution of the thesis. It builds on Chapter 5 and develops a novel network node that is particularly suited to AFE and multi-spectral data sets. It implements a combination of spectral and spatial using hybrid *Morphological-Linear* building blocks. These building blocks can be efficiently implemented on FPGAs and have many properties desirable for AFE.

Chapter 7, Experiments with POOKA: This chapter makes an assessment of techniques developed in the thesis. POOKA is the name given to a 3-layer network of the multi-spectral nodes developed in Chapter 6. This chapter makes a detailed evaluation of the architecture in terms of both implementation using Custom Computers, and quality of AFE algorithms that it can produce. POOKA is compared to a number of more conventional AFE techniques.

Chapter 8, Discussion: This chapter concludes the thesis. The major contributions are summarized and directions of future work are discussed.

Chapter 2

Literature Review

This thesis draws from three fields of study. The first field defines the problem: Automatic Feature Extraction (AFE) in multi-spectral imagery. In Section 2.1 we introduce AFE and briefly describe conventional approaches. Particular attention is paid to linear and non-linear spatial filters.

The second field is the optimization technique: An introduction to Evolutionary Algorithms is given in Sections 2.2.1 through 2.2.3. Application of EA to AFE related optimization problems is described in Section 2.2.4. The motivation for hardware acceleration of EA experiments is presented in Section 2.2.5.

The third field of study concerns the implementation: Field Programmable Gate Arrays are the fundamental hardware building blocks and are described in Section 2.3.1. Their use within custom computers is described in Section 2.3.2. The field of Evolvable Hardware, which combines the fields of EA and FPGAs is described in Section 2.3.3. The main topics relevant to latter chapters of the thesis are summarized in Section 2.4.

2.1 Automatic Feature Extraction in Image Data Sets

Extracting useful, high-level information from images in the presence of noise and uncertain conditions is a process that humans do well but computers find hard. The type of high-level information considered useful depends on the application. Examples of applications include object/target detection and tracking, often used in surveillance and robotic systems, and defect / fault detection often found in automated production systems. A problem of particular interest to this thesis is image or scene classification in remotely sensed imagery. High-speed multi- and hyper-spectral sensors are producing increasing volumes of raw image data. To develop algorithms that exploit multi-spectral data cubes is a time consuming and costly process and therefore automatic feature extraction (AFE) algorithms are an attractive alternative.

In image data sets, AFE is usually decomposed into a processing pipeline with raw image data (taken from a sensor) as input and the desired information as output. Figure 2 depicts this processing pipeline.

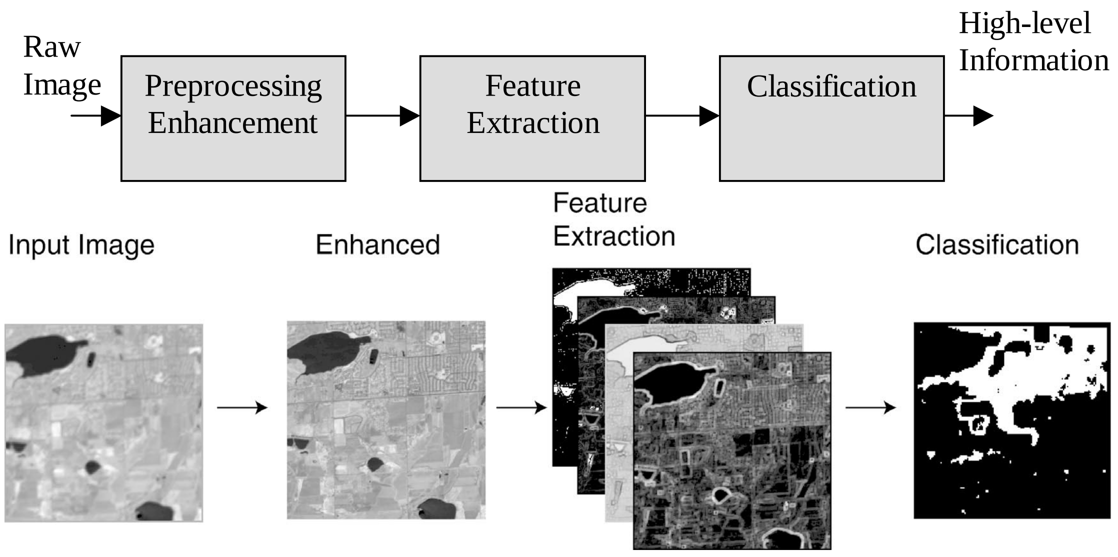


Figure 2: A Typical Image Processing Pipeline

- The first stage of the pipeline is concerned primarily with removing sensor noise and attempts to ‘enhance’ an input image to make subsequent analysis easier. This type of processing is discussed further in Section 2.1.1.
- The second stage then attempts to extract quantities that are the most relevant to a particular problem and essentially derives cues from which high-level decisions can be based. This is known as feature extraction[3] and is discussed in Section 2.1.2.
- The third stage then uses the cues to decide whether regions of interest in the image. This is known as classification and is described in Section 2.1.3.
- Many variants and extensions of this basic pipeline exist. For example many pipelines include a segmentation step [4] where pixels are grouped into coherent objects.

[3] Not to be confused with Automatic Feature Extraction which concerns high-level user defined features of interest.

In multi-spectral data sets, a single scene is imaged at many different wavelengths. This means the image pipeline of Figure 2 is extended to include multiple input images. An example of this extended pipeline is illustrated in Figure 3. In this figure preprocessing is not shown. In the image data sets used in this thesis, significant preprocessing, such as calibration and registration, is performed automatically as the data is collected from the sensor. It can be seen in Figure 2 that the Feature Extraction stage for multi-spectral data sets becomes a multiple-input, multiple-output transformation.

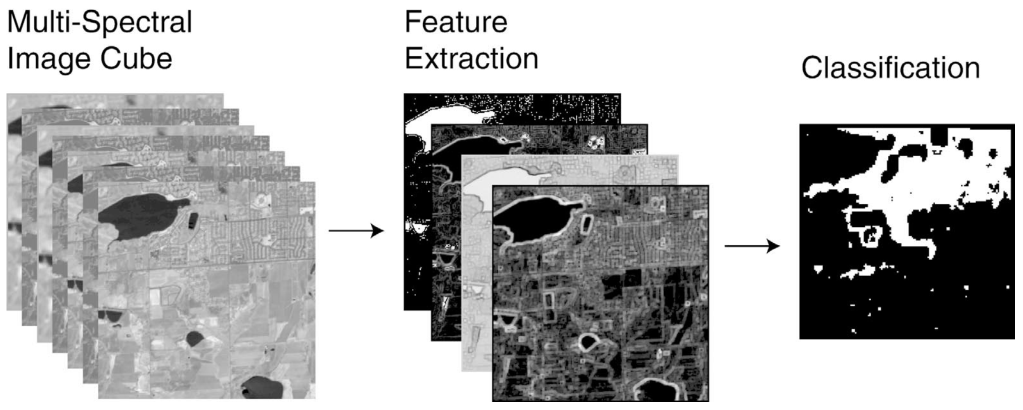


Figure 3: Multi-Spectral Processing Pipeline

2.1.1 Preprocessing and Enhancement

Algorithms used for image enhancement and feature extraction are closely related. Both types of processing attempt to reduce noise and extract useful information for subsequent processing. Traditionally, image enhancement includes point operators, spatial operators, histogram modeling and transform operations [4]. Histogram modeling and transform techniques are applied globally. This means they use information that is calculated by considering all pixels in the image at one time. These algorithms are more expensive to implement in hardware than point and spatial operators, and often require multiple passes through an image. Therefore, they are not considered further in this thesis.

Point operators are applied on pixel-by-pixel basis and include clipping, thresholding, pixel scaling, and range manipulation. Many linear and non-linear (e.g. log, eX) point-to-point transformations can be used. Such basic operations are described in detail in most image processing texts [5], [4].

Spatial filters are a common class of algorithm that is used extensively in image processing and in latter chapters of this thesis. These filters are applied to a finite spatial neighborhood of an image and implement both linear and non-linear functions. Figure 4 illustrates the how the spatial filter is applied for a 3 by 3 neighborhood. The spatial filter is applied to all pixels in parallel. It can be seen that when the filter is applied to edge pixels some of the neighborhood is not defined. One way to deal with this problem is to buffer the image before applying the spatial operator. The valid output image from the filter is reduced in size, relative to the neighborhood size. In the case of the 3 by 3 spatial filter the output image is reduced in size by 2 pixels in both horizontal and vertical dimensions each time the filter is applied.

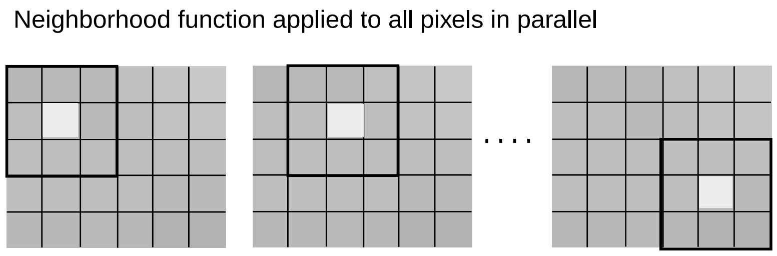


Figure 4: Application of Spatial Filters

2.1.1.1 Linear Spatial Filters

Linear spatial filters apply a multiplicative weight to each pixel in the neighborhood. The weights are defined by a matrix, which is illustrated in Figure 5. In the spatial domain, linear spatial filters represent a convolution of the image with this weight matrix. This is defined in equation 1, where I'(i,j) is the output at pixel (i,j), W(k,l) is the $W_{k,l}^{th}$ weight in Figure 5, and I(i-k, j-l) is the input image at pixel (i-k,j-l). By selecting the appropriate matrix weights, linear spatial filters can implement low-pass, high-pass and band-pass frequency domain filters.

$W_{1,1}$	$W_{1,2}$	$W_{1,3}$
$W_{2,1}$	$W_{2,2}$	$W_{2,3}$
$W_{3,1}$	$W_{3,2}$	$W_{3,3}$

Figure 5: Linear Filter Weight Kernel

$$I^{'}(i,j) = \sum_{(k,l)\in Window}\sum W(k,l)I\ (i-k, j-l) \qquad \textbf{(1)}$$

Low-pass filtering is implemented by using positive weights. This is equivalent to a weighted average of neighborhood pixels and is used widely for image smoothing. Common weight kernels in smoothing include 'flat' kernels, where all weights are set to the same value, illustrated in Figure 6. It is also common to use the weights to approximate a Gaussian function, also illustrated in Figure 6. High pass filters use a kernel with a positive center weight and negative outer weights. An example is illustrated on the right of Figure 6. These filters are used to enhance high frequency components in an image such as edges and fine detail.

Flat Average

1	1	1
1	1	1
1	1	1

Sampled Gaussian

1	4	1
4	12	4
1	4	1

High-Pass Filter

-1	-1	-1
-1	8	-1
-1	-1	-1

Figure 6: Example Weight Matrices

Linear spatial filters also include a large number of gradient operators that are useful in enhancing edges. Weight kernels suggested by Roberts [6] and Sobel [7] are shown in Figure 7. Two weight matrices are associated with each operator and are used to estimate gradient in two orthogonal directions. Weight matrices by Sobel estimate Grad$_x$ and Grad$_y$, but any two orthogonal directions can be used. The magnitude of the total gradient is then usually calculated by application of equation 2.

$$|Grad| = \sqrt{Grad_x^2 + Grad_y^2} \qquad \textbf{(2)}$$

Due to the computational complexity of Equation 2, approximations have been suggested. For example, the absolute value of Grad$_x$ and Grad$_y$ can be calculated and the results simply summed. Another approach, used by Kirsch in [8], is to apply multiple spatial filters at various orientations and then use the maximum value of the responses.

Roberts cross-gradient weights

$$\begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}$$

Sobel Grad_x

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

Sobel Grad_y

$$\begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix}$$

Figure 7: Gradient Weight Matrices

2.1.1.2 Non-Linear Spatial Filters

Linear filters are ideal for suppressing purely Gaussian noise. When noise is non-Gaussian, nonlinear filters can be more effective. For example median filters are ideal for suppressing Laplacian noise [9].

Mathematical Morphology defines a family of non-linear spatial filters that are based on neighborhood order statistics. Early work, that lay foundations for mathematical morphology can be found in [10] and Preston in [11]. Preston's work is of particular note since it describes implementation and use of mathematical morphology for practical applications prior to the theoretical development of the field. Highly mathematical works followed [12], and morphology now has a firm foundation in set theory.

In their simplest form, there are no weights associated with neighborhood samples. The shape of the neighborhood is known as the structuring element or region of support, examples of which can be seen in Figure 8, for circular and cross structuring elements. The structuring element defines the set of pixels from which an order statistic is derived. The fundamental spatial filters in mathematical morphology are erosion and dilation [13]. Erosion in simplest terms is a minimum of pixels, and dilation is the maximum of the pixels in the set defined by the structuring element. Figure 9 illustrates the result of applying these two filters, to binary images. Note, in the binary case a minimum corresponds to a logical AND of neighborhood pixels. For dilation, the maximum corresponds to a logical OR of neighborhood samples. Due to the close relationship between morphology, and logical operations, extremely efficient hardware implementations are possible.

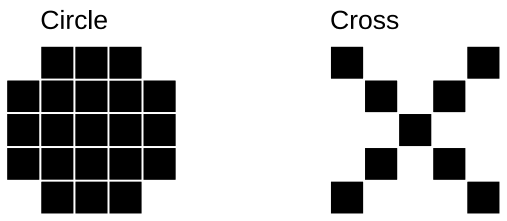


Figure 8: Example of structuring elements in a 5 by 5 Morphological Filter

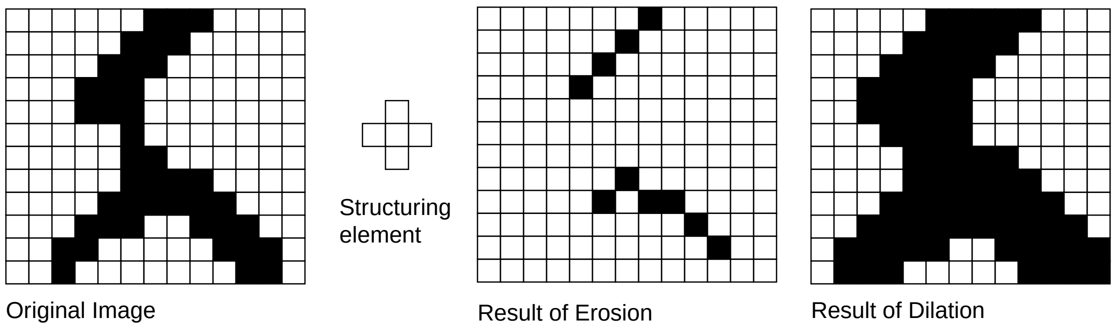


Figure 9: Example of binary Erosion and Dilation.

Erosion and dilation are used widely for low-level image enhancement and smoothing. Particularly useful filters in this regard are summarized in Table 1, which apply erosion and dilation successively in a particular order. Usually the shape of the structuring element is constant between successive erosions and dilations.

Morphological Filter	Notation and Defintion
Erosion	$F = Img \ominus S_t$
Dilation	$F = Img \oplus S_t$
Opening	$F = Img \circ S_t = (Img \ominus S_t) \oplus S_t$
Closing	$F = Img \bullet S_t = (Img \oplus S_t) \ominus S_t$
Open-Close	$F = (Img \circ S_t) \bullet S_t$
Close-Open	$F = (Img \bullet S_t) \circ S_t$

Table 1: Summary of Morphological Smoothing Filters [13]

Filters based on the median are also related to the field of mathematical morphology. In this case the order statistic that is used is the median. The relationship between median filters and mathematical morphology is discussed in [14] and is most easily seen in application to binary images. Consider the 1 dimensional image of Figure 10, where a spatial median is applied with a neighborhood of width 3. In this example the median is implemented with a logical OR (+) of 3 AND (implied) expressions. For gray-value image processing this is equivalent to a maximum of 3 minimums. Each of the 3 minimums is known as a *minterm*.

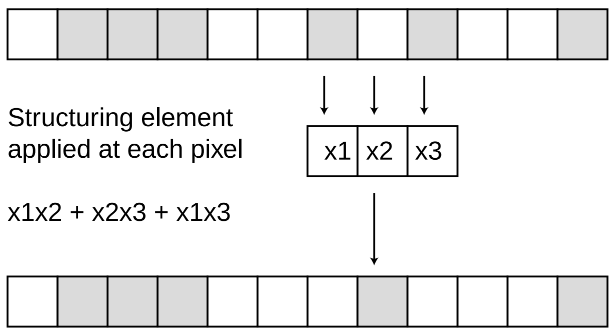


Figure 10: Application of spatial median

Median filtering has been used extensively for image smoothing, particularly when noise is non-Gaussian. The logical expression in Figure 10 is known as a positive Boolean function or PBF. This subset of Boolean logic includes all functions, where negation of input variables is not allowed. The larger set of filters, defined by the family of PBFs, has also received considerable attention for noise removal in both signal and image processing and include weighted median filters [15], weighted order statistic filters [16], and stack filters [9], [17].

A natural approach to deal with the combination of Gaussian and Non-Gaussian noise found in most practical problems, is to define a hybrid system that incorporates both linear are non-linear filtering behavior. Examples of this approach are L-filters [18] where multiple order-statistic filters are combined with a linear combination. In another approach [19], the outputs from a bank of linear phase FIR filters are combined with a median or order statistic filter. A comprehensive account of nonlinear filters, particularly with respect to hybrid approaches can be found in [20]

2.1.1.3 Preprocessing in Multi-Spectral Data Sets

Preprocessing in multi-spectral data sets includes a large body of work that is outside the scope of this thesis. Algorithms are often designed on a sensor by sensor basis and include correction of systematic defects and undesirable sensor characteristics, as well as calibration of sensor and atmospheric models [21]. Such processing is often performed as the data is collected, and is essential to subsequent data analysis. This thesis is concerned with the latter stages of multi-spectral data processing. One type of preprocessing that is worth note is data reduction techniques. Multi and Hyper spectral sensors produce data cubes with large spectral dimensions. Sometimes it is inefficient to apply algorithms directly to these large dimensional data sets, and therefore transformations such as Principle Component Analysis [21] and the Minimum Noise Fraction [22] are applied. These transformations reduce the dimension of the data by minimizing the correlation, and therefore redundancy, between spectral channels.

2.1.2 Feature Extraction

Feature extraction attempts to find measures that will separate points of interest from the background. This stage of the pipeline can be considered a transformation from the image space, where each pixel usually represents intensity, to a feature space where pixels represent more abstract quantities. In simplest terms, these quantities often represent things such as smoothness or roughness, but can vary greatly depending on the particular feature of interest. While the usefulness of a particular measure depends on the problem, there are general characteristics that are often desirable. These include:

- Rotationally invariant: It is often desirable that AFE algorithms perform equally well, regardless of the orientation of the image. This implies measures derived in feature extraction should be rotationally invariant.
- Insensitive to changes in illumination: Ideally, a feature of interest should be discernable in many different lighting conditions. Usually illumination will

affect the magnitude of all pixels in the same way. Algorithms that depend on particular pixel magnitudes must be carefully considered.

- Scale invariant: Ideally, a good feature extraction stage will perform equally well regardless of the scale of the input image.

Texture classification and characterization is an important part of many image-processing applications and has been studied for a number of years. One of the earliest applications was terrain classification in remotely sensed images, which is of particular relevance to this thesis. In terms of feature extraction, texture is difficult to quantify. Texture has been characterized statistically, structurally and spectrally. Spectral methods are based in the frequency domain and are not considered. An excellent review of early statistical and structural techniques is presented in [23].

Haralick in [23] suggests texture measures based on the spatial gray-tone dependence matrix. This matrix consists of relative frequencies P(i, j), where i and j are two particular gray values that occur a distance d, at direction θ from one another throughout the image. The dependence matrix is formed through inspecting the image. This must be performed over a sufficiently large area to make the frequency estimates meaningful. Also for this reason, image gray values are usually quantized, typically 8 or 16 levels. Typically 2 to 10 different values for distance and 4 to 16 directions are used. For each distance and angle, Haralick defines a set of 32 statistical features, which can be calculated from the dependence matrix. In [24] Haralick used these measures for an AFE application in satellite images. A problem with the dependence matrix approach is computation time. Multiple matrices must be formed for various distances and angles before measures can be derived. This can be expensive to implement in hardware.

2.1.2.1 Linear Spatial Filters

The spatial filters described in Section 2.1.1 have also been used extensively for feature extraction. Edges play an important role in discriminating texture and therefore the gradient operators of the previous section are often used. Laws in [25] suggested several 3 by 3 and 5 by 5 weight matrices specifically for texture. Some of these weight matrices, known as “texture energy measures” are illustrated in Figure 11.

The Laws texture measures implement band-pass filters in the frequency domain. It can be seen in Figure 11, that some of the masks are similar to smoothing and edge detector masks encountered earlier. Another important point to note is that masks can be asymmetric. This increases the operators' ability to discriminate textures, but multiple filters are required to make the discrimination rotationally invariant. In practice a bank of filters, with multiple rotated versions of the mask are applied. It should also be noted that many Laws masks have zero sum. Laws suggested the sum of squares, or sum of absolute values were therefore the most useful measures.

1	2	1
2	4	2
1	2	1

-1	-2	-1
2	4	2
-1	-2	-1

-1	0	1
-2	0	2
-1	0	1

1	-2	1
-2	4	-2
1	-2	1

-1	2	-1
-2	4	-2
-1	2	-1

-1	-2	-1
0	0	0
1	2	1

1	0	-1
0	0	0
-1	0	1

1	0	-1
-2	0	2
1	0	-1

1	-2	1
0	0	0
-1	2	-1

Figure 11: Examples of Laws 3 by 3 Texture Measures

Interesting extension to Laws work is described in [26]. This work asked if it the specific weights used by Law were important or simply the general structure of the spatial filter. They proposed several modifications and concluded the latter: that Laws spatial filters can be considered a combination of concentric spot masks and center weighted edge detectors.

A large body of more recent work in statistical texture characterization has focused on Gabor filters [27]. Gabor filters are Gaussian shaped band-pass filters, centered at a particular frequency. Similar to Laws texture masks, the Gabor filter is often asymmetric. Therefore, a bank of Gabor filters are usually implemented, each tuned to specific frequencies and orientations.

Many papers, comparing texture measures have been published [28], [29], [30] and [31]. Results seem problem dependent and no single technique consistently outperforms others.

2.1.2.2 Non-linear Spatial Filters

Mathematical morphology can be considered a structural or geometric approach to feature extraction and texture segmentation. Traditionally structural approaches attempt to characterize texture through a decomposition of the image into a number of textural primitives. Once these primitives have been identified, statistical measures can be derived or syntactic rules found for their placement [32]. Mathematical morphology can be used to characterize shape and size of textural primitives through selection of appropriate structuring elements and therefore can be considered a structural approach.

The morphological equivalent of gradient operators discussed in Section 2.1.1 is the morphological range defined in equation 3. Img is the original image, S_t refers to a particular structuring element and $\oplus$ and Θ refer to morphological dilation and erosion respectively.

$$\text{Range} = (\text{Img} \oplus S_t) - (\text{Img}\ \Theta\ S_t) \tag{3}$$

Morphological filters are particularly effective for feature extraction when the feature of interest has unique size or shape compared to the rest of the image. For example, if the features of interest in a remotely sensed image are the roads, they could be characterized by a linear structuring element. In two-dimensional images, a linear structuring element is not rotationally invariant. Therefore, multiple rotations of the structuring element are usually applied. This is similar to the filter bank approach described previously for linear filters. In the morphological case, the maximum or the minimum response from the multiple filter rotations is used. The maximum is used if the filter is dilation, or opening. The minimum is used if the filter bank implements erosion or closing. The multiple rotations in a filter bank for a linear structuring element are illustrated in Figure 12.

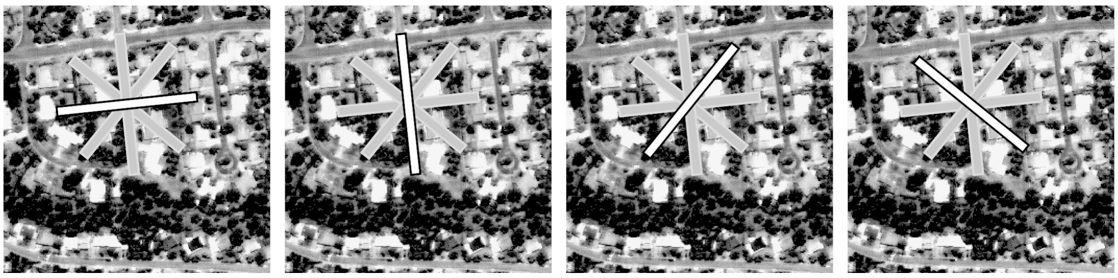


Figure 12: Using multiple rotations of asymmetric structuring elements

When the feature of interest has unique size relative to the rest of the image, morphological filters can be tuned to extract the feature of interest. For example in closing, dark objects that fit within the tuned structuring element will be removed from the image. Opening will have the same effect for bright objects in the image. Often the feature of interest happens to be information that is removed from the image. This information is known as the residule. Image subtraction is then used for recovering the residule. A simple example of this image subtraction is seen in the top-hot transform. Equation 4 describes this operator where $\circ$ corresponds to morphological opening and S_t is a particular structuring element.

$$\text{Top-hat} = \text{Img} - (\text{Img} \circ S_t) \qquad \textbf{(4)}$$

The idea of using the shape and size of the structuring element to sieve an image of particular information is extended in the field of granulometries, first proposed in [33]. The term Pattern Spectrum is also used to describe a particular granulometric approach. A comprehensive discussion of Pattern Spectrum is given by Maragos in [34]. Explanation of granulometries to come is based on works by Dougherty in [35] and [36].

The granulometric approach, involves opening or closing an image with successively larger structuring elements. The notation kS_t, is often used to denote a particular size k of a structuring element via equation 5.

$$kS_t = \overbrace{S_t \oplus S_t \oplus \dots \oplus S_t}^{\text{k times}} \quad \textbf{(5)}$$

An important property of granulometries comes from the fact that for $k_2 > k_1 > 0$, opening by k_2: $Img \circ k_2S_t$ will be a subset of the opened image by k_1: $Img \circ k_1S_t$. This can be most easily understood in terms of the sieving analogy. More of the image will be smoothed, or removed by the k_2S_t structuring element than with k_1S_t.

$\Omega(k)$ defines the total area of the image that is left after the $Img \circ kS_t$ opening is applied. $\Omega(0)$ defines the area of the original area, and can be calculated by summing all pixel values. The property described above means that $\Omega(k)$ is a decreasing function of k an that $\Omega(k) = 0$ for sufficiently large k. $\Omega(k)$ is known as the size distribution, and a normalized size distribution is defined by Equation 6.

$$\Phi(k) = 1 - \frac{\Omega(k)}{\Omega(0)} \quad \textbf{(6)}$$

$\Phi(k)$ is an increasing function from 0 to 1 and can be interpreted as a probability distribution function. The derivative of this function represents a probability density function and is calculated by Equation 7. This function is known as the pattern spectrum.

$$d\Phi(k) = \Phi(k+1) - \Phi(k) \quad \textbf{(7)}$$

Pattern spectrums have been used extensively for texture and object characterization in image data sets. Classification has been applied directly to pixel pattern spectrums in [37]. It is also common to use calculate statistical moments of the pattern spectrum prior to classification. Work by Aubert et al. [38] is of particular interest. The computational cost of computing granulometries via opening and closing leads the authors to suggest *pseudo-granulometry* based on erosion and dilation respectively.

2.1.2.3 Feature Extraction in Multi-Spectral Data Sets

There are several algorithms that are used in multi-spectral feature extraction worth particular note. Band ratios are an arithmetic division of two spectral bands. Many extensions to this ratio such as Normalized Differential Vegetation Index (NDVI) have been described [21]. The NDVI is shown Equation 8, $S_1(i,j)$ and $S_2(i,j)$ refer to two different spectral channels that are taken as input to produce $S_{NDVI}(i,j)$. This can be seen as a type of multi-spectral point operator.

$$S'_{NDVI}(i,j) = {S_1(i,j) - S_2(i,j)} \Big/ {S_1(i,j) + S_2(i,j)} \qquad \textbf{(8)}$$

Another important multi-spectral technique is a weighted sum of spectral channels described by Equation 9 where N is the number of spectral channels being considered. In this case the weight vector, W_n is often referred to as a spectral signature and the dot product of spectral channels and the weight vector can be considered a spectral distance to the particular signature. This type of processing is considered further in the next Section on Classification.

$$S'(i,j) = \sum_{n=1}^{N} W_n * S_n(i,j) \qquad \textbf{(9)}$$

2.1.3 Classification

The 3rd stage in the processing pipeline of Figure 2 is classification. The measures derived in feature extraction are used to decide whether particular pixels belong to a target class or are of interest. This stage is most easily understood in feature space. Usually a number of measures, N are derived in feature extraction. Feature space is then the N-dimensional histogram of these measures. This is illustrated in Figure 13 for N=2.

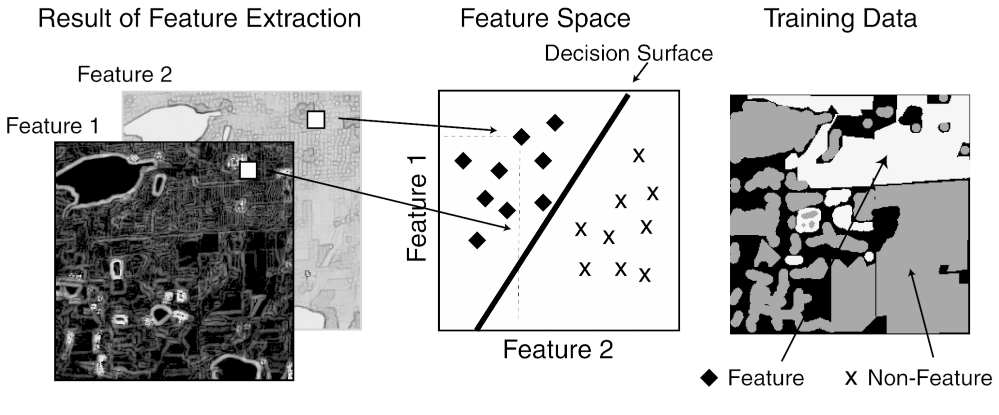


Figure 13: Feature Space

Classification is concerned with dividing feature space, by drawing boundaries known as decision surfaces, into a number of classes. In this thesis, only 2-class problems are considered. This means a pixel belongs to one of two classes: one class usually corresponds to a *feature of interest* and the other class, *non-feature*. Figure 13 illustrates a common decision surface used for two-class problems. A line in 2 dimensions, or hyper-plane in higher dimensions, is used to divide feature space into the two areas that correspond to the two classes.

Classification can be considered supervised or unsupervised. This thesis deals exclusively with supervised learning. This means training data is supplied so that decision surfaces can be optimized for a particular problem. Training data consists of a collection of pixels that have been pre-labeled as belonging to a particular class. This is referred to as a target classification and an example can be seen on the right of Figure 13. White corresponds to *feature of interest* and gray corresponds to *non-feature*. Black is unlabelled and can be considered *don't care* (or more usually *don't know*). The supervised learning approach uses the target classification to calculate a classification error. The decision surface can then be optimized to minimize this error.

2.1.3.1 Statistical Classification

Pattern classification has firm foundation in statistics and probability. Only a brief introduction to this large field of research is possible. Comprehensive introductions to these topics are available in many standard texts [39], [40], and [41].

Using the statistical approach, the probability of misclassification is minimized by estimating the Bayes posterior probabilities $P(C_k \mid X')$. This is the probability of a pattern belonging to class C_k when X' is observed. The pattern is then assigned to the class with the largest posterior probability. For the two class problem, if $P(C_0 \mid X') > P(C_1 \mid X')$ then X' is assigned to class C_0. A larger number of techniques have been proposed which differ in the assumptions made in modeling the probability density functions of the two classes. Usually Gaussian distributions are assumed, which are characterized by the distribution mean and covariance matrix.

In the simplest case the covariance matrices are assumed equal for all classes. Additionally, if all features (dimensions in feature space) are assumed independent, the classifier is known as the Minimum Distance Classifier. In this classifier, posterior probabilities can be estimated by a Euclidean distance between observed patterns X', and the class mean. During training, the class mean (often referred to as a prototype vector) is estimated from the training data. In application, a pattern is assigned to the closest class based on distance to the prototype vectors. For the two-class problem, the decision surface is a hyper-plane perpendicular to a line segment joining the two class prototypes. Spectral Angle Mapper (SAM) is a variant of Minimum Distance that is used in experiments latter in the thesis. In this case, the magnitude of pattern vectors is normalized prior to classification. SAM is used in remote sensing communities and is often applied directly to the spectral dimension of a multi-spectral image [42]. Normalization is motivated by the need to produce an illumination insensitive classification. By normalizing the magnitude of pattern vectors, variations in the relative quantities of spectral channels is more easily discerned.

When class distributions assumptions are relaxed, more complex decision surfaces can be made. In the Maximum Likelihood Classifier, no assumptions are made concerning the class covariance matrices. For the two-class problem, this leads to decision surfaces based on *hyperquadrics.* The Maximum Likelihood is used widely in remote sensing data sets [43] and latter in the thesis.

2.1.3.2 Neural Networks

Neural networks have been used extensively for pattern recognition problems and are particularly relevant to architectures considered in this thesis. The Perceptron is a linear thresholding element fundamental to the field of neural networks.

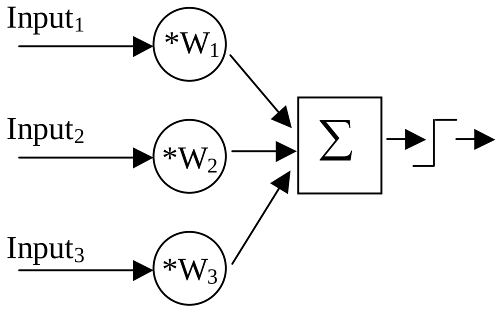


Figure 14: The Perceptron

The Perceptron was suggested by Rosenblatt in [44] and is depicted in Figure 14. It implements a hyper-plane decision surface, similar to that produced by the Minimum Distance Classifier. More complex decision surfaces can be formed by combining linear thresholding elements in a multiple layer, feed-forward network. A key work to the development of multiple layer neural networks was a supervised learning algorithm [45]. This work introduced a gradient descent method known as Back Propagation. Necessary to this training algorithm is the use of differentiable sigmoid activation functions. The use of sigmoid activation functions also means the network output is an estimate of the posterior class probabilities [41]. Multiple layer neural networks are able to produce arbitrary decision surfaces examples of which can be seen in Figure 15. This illustration is based on work by Lippmann [46], which provides an excellent introduction to the early work on neural networks.

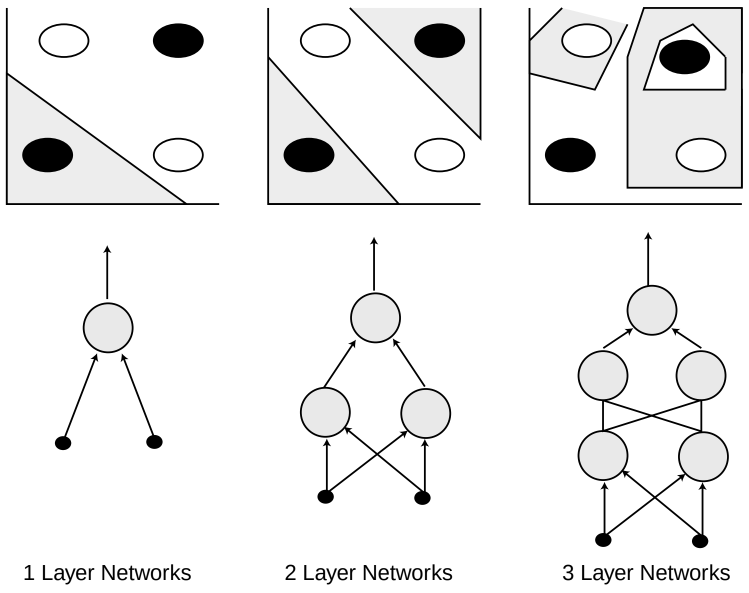


Figure 15: Multiple Layer Network Decision Surfaces

When a linear perceptron is applied to a spatial neighborhood (without thresholding), the function it implements is a convolution, or linear spatial filter described in Section 2.1.1. This type of network is known as a convolutional neural network. One of the first implementations of convolutional neural networks was called the neocognitron and was reported in [47]. Subsequently, they have been applied to a wide range of problems in image and signal processing [48].

2.1.4 Discussion

Section 2.1 has provided a brief overview of constituent parts of a typical AFE algorithm, as well as the more specific techniques referenced in the thesis. While the decomposition of AFE into preprocessing, feature extraction and classification is seen widely in the literature, in practice these stages are closely tied.

An important consideration in supervised AFE is generalization. Obtaining an accurate classification on the training data is important, but it is the application of optimized algorithms to out-of-training data that is the primary motivation. This is a much more difficult problem to quantify. In more general terms, the interplay of pipeline stages can potentially affect the generalization ability of a particular AFE algorithm. For example, classification techniques such as the maximum likelihood are capable of forming complex decision boundaries and therefore could be applied directly to the raw data and achieve low classification error on the training data. In practice, this often leads to poor performance on out-of-sample data and therefore preprocessing and feature extraction are usually preferred. A good set of features will make classification easier and hopefully lead to good generalization.

Deciding what features will lead to robust AFE is known as Feature Selection and is discussed further in the Evolutionary Algorithms section. One approach to this problem of particular interest is the co-optimization of feature extraction and classification components. This means feature selection can be directed towards easily classifiable subsets, while simultaneously leading to simpler classifiers. An example of this approach was presented in [49], where a Morphological shared-weight neural network, capable of both feature extraction and classification was used for automatic target recognition problems.

A problem with the co-optimization approach is that learning algorithms become more complex. There are a large number of potentially useful transformations that could be used for feature extraction and optimization soon becomes intractable. A potential solution to this problem lies in Evolutionary Algorithms, which are the topic of the next section.

2.2 Optimization with Evolutionary Algorithms

Evolutionary Algorithms (EA), define a family of stochastic search and optimization techniques that include genetic algorithms, genetic programming, evolutionary programming and evolutionary strategies [50]. Evolutionary algorithms are one of the most flexible optimization techniques in use today. For this reason EA have been applied to a variety of research, industrial and commercial problem solving activities [51].

It is often convenient to visualize a fitness landscape depicted in Figure 16. This landscape covers all possible configurations of a particular model with two degrees of freedom. Fitness is an abstract measure of how well a configuration solves a particular problem. It can be seen in Figure 16, that this is a hard optimization problem with many local optima, and possibly several near-global optima. EAs are a computationally intensive search technique that can find global optima in these types of problems. They do this by maintaining a number of search paths in parallel. This is known as a population. The population provides a wide sampling of the fitness landscape, which is important in avoiding local optima. The population of search paths, or chromosomes, is then manipulated from one generation to the next in an effort to find the global optima.

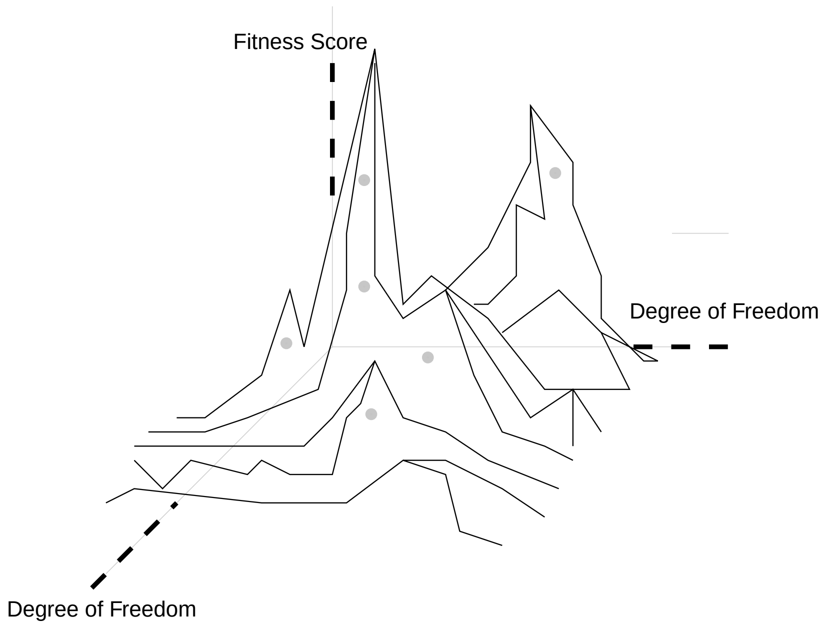


Figure 16: A Fitness Landscape

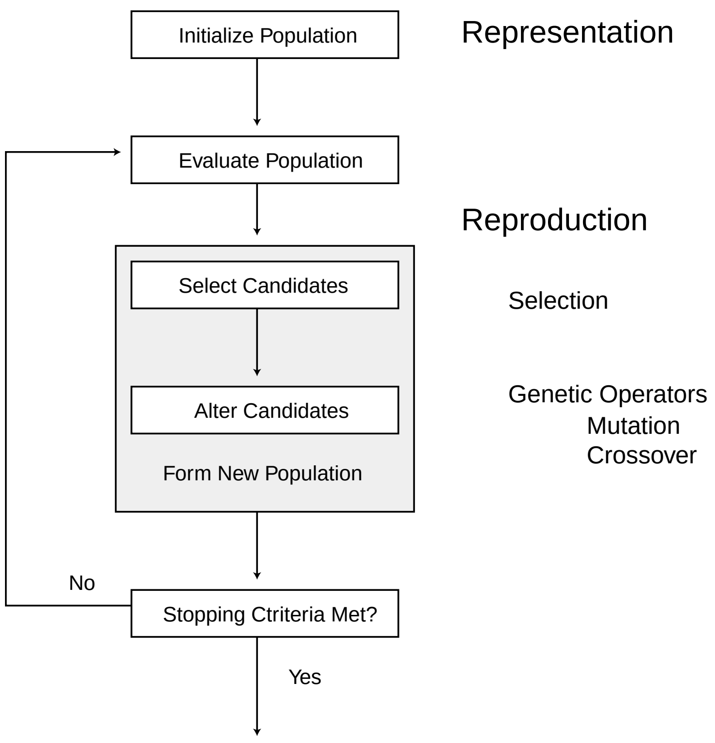


Figure 17: The general Evolutionary Algorithm

A general procedure for evolutionary algorithms can be seen in Figure 17. The initial sampling of the search space that generates the initial population is usually random. The chromosomes, or candidate solutions within the population are then manipulated from one generation to the next in reproduction. Reproduction is usually based on fitness, so that chromosomes that solve the problem well have more chance of being present in the next generation. After, a finite number of generations the population that results will hopefully contain highly fit chromosomes and therefore good solutions to the problem. There are a great many ways EA can be applied to particular problems. The most common variation in EAs occurs in:

- Representation: How candidate solutions within a population are represented. The representation effectively defines the degrees of freedom or search space for a particular problem. It is therefore often problem specific.
- Selection: How the new population is selected from the old population.
- Genetic Operators: How the population is altered from one generation to the next.

Some of the more common variants will now be discussed. Readers familiar with EA can move to Section 2.2.4 where the application of EA to AFE problems is described.

2.2.1 Representation

One of the first EA strategies to be introduced was Genetic Algorithms by Holland [52]. In this work bit strings are used to represent individuals within the population. The advantage of the bit-string representation is that they are universal. Bit strings are a 'general purpose' representation that can be applied to a variety of problems. They also allow more theoretical investigations of Evolutionary Algorithms to be made. This is discussed more in terms of genetic operators. The problem with bit string chromosomes is that they are not always the most natural or simplest way to represent a problem. An alternative bit string representation, termed Messy GAs was proposed in [53]. It represents chromosomes with variable length bit strings. This increased flexibility is often convenient for many types of problems. For parameter optimization problems, where precision is often crucial, floating point numbers have been used successfully. Floating-point representations were first used and developed in the field of Evolutionary Strategies. A comprehensive account of early work in this type of EA is given in [54]. In other applications such as combinatorial optimization problems, integers or natural numbers are often more appropriate.

Genetic Programming is a type of evolutionary algorithm first described by Koza in [55]. Chromosomes are represented by high level, often problem domain specific, hierarchical expressions similar to software programs. Koza’s initial structures were based on Lisp S-expressions, but the principles have now been applied to a variety of programming languages including C, C++, Pascal, FORTRAN and Small talk. An example of a Genetic Programming chromosome is shown in Figure 18.

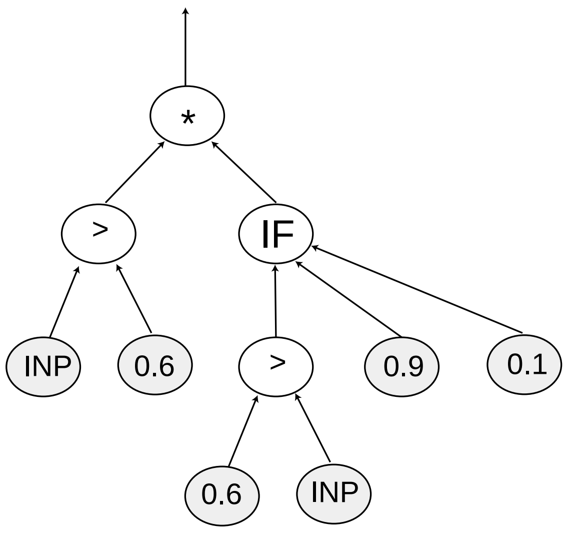


Figure 18: A Typical Chromosome in Genetic Programming

Trees are composed of genetic material from:

- A population of terminals (gray): constants and input variables.
- A population of functions: These include arithmetic functions, programming operations such as IF and FOR constructs, logical and Boolean functions as well as problem domain specific functions.

A requirement known as *closure* is often introduced which means that all functions within a population are able to receive the output of any other function or terminal as input. An initial population is usually chosen at random and the genetic operators of reproduction and crossover are applied at each generation. Individuals are variable length and expand and contract as the evolutionary algorithm progresses.

2.2.2 Genetic Operators

The success of EA in reaching global optima depends largely on how effective genetic operators are at generating fitter individuals. Due to the problem specific nature of representation, genetic operators are also often representation dependent and therefore a large number of strategies have been suggested. In this section, the basic principles and motivation of the two fundamental EA operators: *mutation* and *crossover* are described.

The mutation operator is applied to a single parent chosen from the population and provides the EA with noise or randomness that is required to explore the search space. In the bit-string representation, mutation is applied by random bit flips at a particular probability. In other representations more complex, problem dependent mutation schemes are used. Methods developed in Evolutionary Strategies usually implement mutation rates that vary during the course of evolution [54]. This means the search space can be explored uniformly at first with high probability of mutation and then locally towards the end of evolution with smaller probabilities.

The second genetic operator is crossover and is usually applied to two parents chosen from the population. Single point crossover is used to describe the process. Two parent chromosomes are selected: C_1 and C_2 in Figure 19. A crossover point is randomly chosen and the parent sub-strings are then swapped. If an individual containing sub-string C_1S_1 and an individual containing sub-string C_2S_2 are both fit individuals, then an offspring containing both sub-strings C_1S_1 and C_2S_2 may also be a fit if not fitter individual. The two new chromosomes produced by this sub-string swap are often called children or offspring.

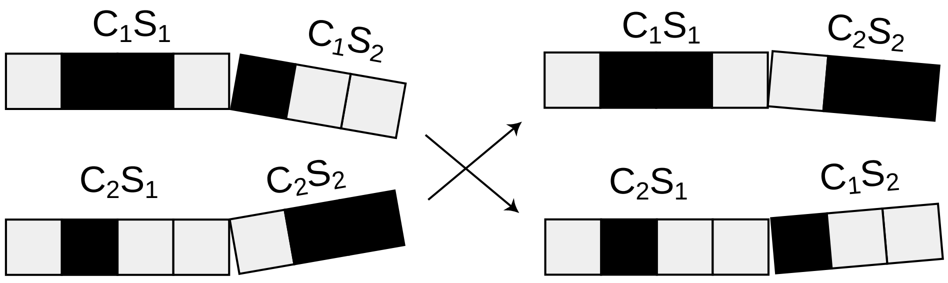


Figure 19: The One-Point Crossover Operator

The idea of sub-strings or building blocks is considered further in theoretical investigations of EA search techniques. The term *Schemata* refers to a sub-string or set of sub-strings found in a population of individuals. Interesting insights are gained by considering evolution of these chromosome sub-strings rather than the chromosomes themselves:

Schema Theorem or Building Block Hypothesis: Short, low-order, above average schemata receive exponentially increasing trials in subsequent generations of a genetic algorithm [52].

Another type of crossover that is useful in visualizing the building block hypothesis is seen in Genetic Programming [55]. In this case the representation is the program like tree that was illustrated in Figure 18. Two parents are chosen, and a node randomly selected from each. The sub-tree formed by all nodes and connectivity below the selected nodes are then swapped between individuals. An extension to this approach was introduced [56] called automatically defined functions (ADF). This approach attempts to dynamically protect sub-trees that are useful building blocks for solving the problem. A particular sub-tree that is deemed to be useful is replaced by a single node in the chromosome and is therefore protected from further modification by crossover. This is useful for optimizing complex systems and can take advantage of regularities, similarities and patterns within a problem domain.

2.2.3 Selection

Having discussed the Genetic Operators, the question remains, how are the parent chromosomes selected from the population. The most straightforward approaches are fitness-proportionate techniques where the probability of an individual being selected is proportional to its fitness. The *roulette wheel* technique was the first method proposed to implement this [57]. Many other selection techniques have now been suggested. These are partly inspired by two problems encountered using the roulette wheel technique:

- When the difference in fitness between individuals is small, there is little selection pressure for fit individuals since they are hard to differentiate.
- When one individual has an unusually high fitness compared to the rest of the population (often called a super individual), the individual may dominate the population leading to premature convergence.

In this thesis, a ranking technique known as *Elitism* is used. This technique uses the relative orderings of individual fitness values rather than the fitness values themselves. Rank-based selection methods can therefore differentiate between close scoring individuals. Other selection methods include tournament selection, steady state GA and island models. Good introductions to the large number of Selection strategies can be found in [57], [50], [58].

The Elitist Selection scheme used in this thesis implements a predetermined schedule for reproduction. This approach means the designer must supply more parameters, but also means the designer has greater control. An example of a schedule is shown in Figure 20. A fixed number, or certain percentage, of fit individuals are called the elite and are carried through into the next generation without alteration. In Figure 20, the top 10 individuals from a population of 100 copied directly to the new population.

EA Parameter	**Number**
Population Size	100
Parents	80
Number of Generations	50
Reproduction	
Elite	10
Mutated Elite	20
Crossover (parents chosen from top 80%)	30
Mutation (parents chosen from top 80%)	30
Random Generation	10

Figure 20: Selection Schedule for Rank Selection

Another proportion of the population is then generated by mutation of a chromosome randomly chosen from the elite. This is to encourage a hill-climbing type search in good chromosomes. In Figure 20, there are 20 individuals generated by mutating the top 10. Another, generally larger proportion of the population is generated by crossover and mutation of randomly selected parents. This selection is not usually based on fitness, but again a percentage of the population. For example in Figure 20 the value of *Parents* dictates selection from the fittest 80 chromosomes.

It is also common for a schedule to include offspring through mutation alone. In Figure 20, another 30 individuals are generated by mutation of parents randomly selected from the fittest 80 individuals. The final 10 individuals that form the new population of 100 are generated through random re-sampling of the search space. While it is hoped the EA can outperform random search, it is sometimes helpful to include the potential of new building blocks at each generation.

An important part of selection is to maintain diversity in the population so that the EA does not prematurely converge to a local minimum. For this reason, the idea of specialization, or niches, appears often in EA literature. Methods of encouraging specialization in a single population have been proposed [59], [60]. Another approach is to use multiple populations. The *island* model of EA [61] is a multiple population technique that has received considerable attention. A number of populations are maintained independently, although there is often some degree of *inter-island* recombination. With the island approach, populations are competing to find good solutions, but since a number of populations are kept semi-isolated premature convergence can be reduced.

2.2.4 Evolutionary Algorithms and AFE

EA have been applied to almost every part of the AFE pipeline. They are particularly useful in optimizing non-linear filters, where it has been traditionally difficult to develop learning algorithms. Examples of this type of work can be found in [62] for morphological filters and [63] for stack filters. Application of EA to feature extraction include optimization of Gabor filters [64], cross-correlation statistics in stereo matching algorithms [65] and edge detection in [66].

Work in [67] describes the evolution of logic trees to perform edge detection in cellular arrays. This is most similar to work that will be described in Chapter 4. An interesting work in [68] applies Genetic Programming to image processing operator design. This work uses a similar approach to the GENIE system to be discussed. EA building blocks are combination of linear and nonlinear filters including Gabor filters, Gaussian and derivatives of Gaussian spatial filters. Non-linear filters include square root, square point operators and the median spatial filter. Discussion of the GENIE system to come will provide readers with a clearer idea of how this kind of approach can be implemented.

2.2.4.1 Evolutionary Neural Networks

Later in the thesis, EA are used to optimize network architectures for AFE problems. EA have been applied extensively to neural network design and optimization. The similarity in architectures between neural networks and the networks developed in this thesis warrants a brief account of this work. EA have been applied to neural networks in a number of different ways:

- Linear chromosomes to represent network weights of a fixed topology [69], [70].
- The chromosome encodes structure as well as weights. The topology can be optimized for a particular problem [71].
- Developmental encoding: The chromosome is a linear program, whose instructions dictate placement and connectivity of network nodes [72].

Specialization is a necessity when trying to optimize large, complex structures such as neural networks, and has slightly different meaning to previous discussions regarding diversity. The idea is to decompose, or modularize the optimization problem and find a collection of sub-components that work well together. Competition within the population, the main evolutionary pressure in traditional EA, is therefore not the only factor. Since sub-components in a network are dependent, and only contribute partially to a complete solution, collaboration is also required.

Incremental Learning is one method of encouraging collaboration. In the incremental optimization of neural networks described in [73], and [74], the optimization begins with one network node. Once this node has reached a specified level of fitness, or there is no improvement in fitness after a specified number of generations, another node is introduced. In some cases, the first node is fixed and the EA is only applied to the second node. In other cases, the chromosome is extended to include both nodes are then evolution continues as normal. This process continues until the network has reached a desired fitness or maximum number of nodes.

Several other mechanisms to encourage collaboration in neural network nodes have been suggested. In [75] multiple populations are used, each one associated with a neuron in a pre-defined network configuration. Neurons are randomly selected from each population, combined in the network and fitness evaluated. The fitness of the network directs evolution within each population, as well as a second abstract population of *blueprints*. Blueprints are essentially records of good neuron combinations. This second population means neurons that participated in high fitness networks, but then were also involved in low fitness networks, are not immediately lost from the gene pool.

A good review of evolutionary neural networks can be found in [76]. Further discussion of these techniques, particularly *Incremental Learning* is made in Chapter 7 with respect to the POOKA network architecture.

2.2.4.2 EA applied to Multi-Spectral AFE and the GENIE system

Relevant to this thesis are works that apply EA to scene classification in remotely sensed data. Daida in [77], uses a genetic programming approach for the classification aspect of the problem. He used a pre-defined set of features based on Haralick's texture measures described in Section 2.1.2. EA was used to optimize a decision tree, based on the textures, to label the feature of interest. Another interesting work that applies EA to automatic scene labeling for multi-spectral images can be found in [78].

Concurrent to this thesis, a software system known as GENIE was developed at Los Alamos National Laboratory [79]. GENE uses a combination of Genetic Algorithm and Fisher Linear Discriminant [41] (hyper-plane classifier with a particular criteria) to classify each pixel in an image as either feature (TRUE) or non-feature (FALSE).

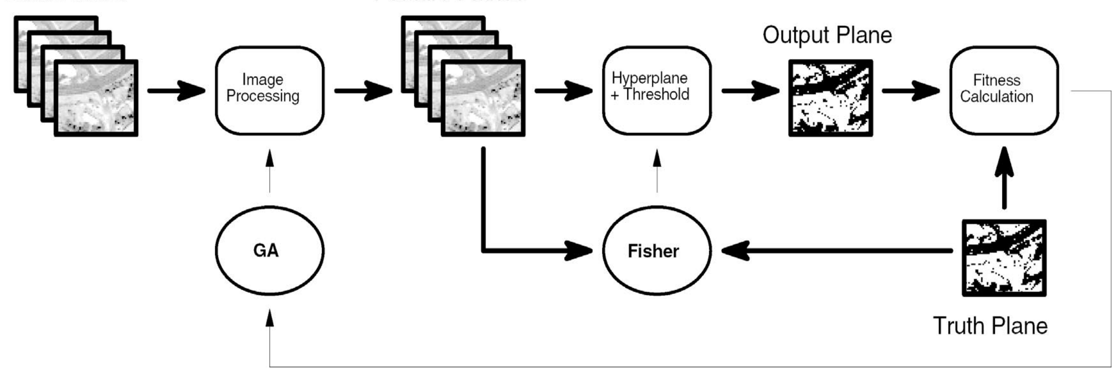
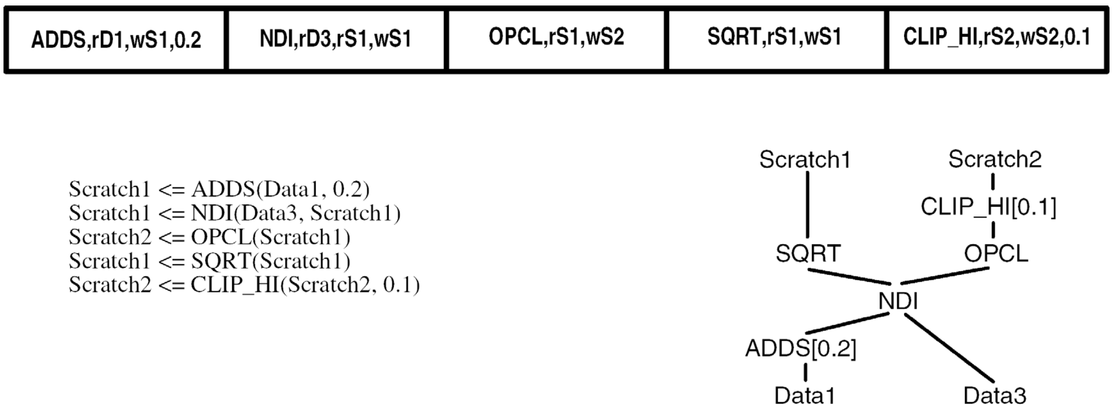


Figure 21: Overview of GENIE architecture

Figure 21 illustrates the high-level architecture. The GA is used optimize image processing algorithms that perform the feature extraction stage. The chromosome therefore specifies a particular transformation from raw *Data Planes* to a set of *Feature Planes.* The linear discriminant is then applied to the *Feature Planes,* which produces a single *Output Plane.* The fitness of the image processing algorithms is found by calculating a distance between the *Output Plane* to the target classification (*Truth Plane* in Figure 21).

Figure 22 illustrates the GENIE chromosome (top) as well as two equivalent representations, which are useful in understanding how chromosomes behave.



```
Scratch1 <= ADDS(Data1, 0.2)
Scratch1 <= NDI(Data3, Scratch1)
Scratch2 <= OPCL(Scratch1)
Scratch1 <= SQRT(Scratch1)
Scratch2 <= CLIP_HI(Scratch2, 0.1)
```

Figure 22: Chromosome representation in the GENIE system [80]

GENIE encodes a linear string chromosome. Each chromosome is composed of a number of genes illustrated at the top of Figure 22. Each gene specifies a particular image processing operation, as well as its connectivity to other genes. Each image processing operator has one or more inputs, performs some predefined computation, and one or more outputs. Table 2 summarizes a subset of the image processing operators that are available to the GENIE system. The lines of code on the left of Figure 22, shows how the chromosome is translated for execution. Scratch planes can be considered temporary images. Genes write their output to scratch planes, which are then used as input by genes latter in the chromosome. The interaction of genes and scratch planes can be visualized most easily as a graph, illustrated on the right of Figure 22. The graph has multiple inputs, taken from the *Data Planes* and multiple outputs (Scratch1 and Scratch2), which are used as *Feature Planes,* and input to the linear discriminate.

Gene Code	Operator Description	Gene Code	Operator Description
ADDP	Add two images	SOBEL	Sobel Gradient
SUBP	Subtract two images	LAW	Neighborhood texture measure
ADDS	Add a scalar to an image	SD	Neighborhood standard deviation
SUBS	Subtract scalar from image	EROD	Morphological Erosion
MULTP	Multiply two images	DIL	Morphological Dilate
DIVP	Divide two images	OPEN	Morphological Open
MULTS	Multiply image by scalar	CLOSE	Morphological Close
DIVS	Divide image by scalar	OPCL	Morphological open-close
SQRT	Square-root of image	CLOP	Morphological close-open
LINCOMB	Linear combination of images		

Table 2: Sample of GENIEs image processing operator pool

The GENIE system uses a generational GA with elitism that was discussed previously. Single-point crossover is applied to the string representation of Figure 22, at gene boundaries. Three types of mutation are implemented:

1. Gene paramaters: Many image processing operators have paramaters such as scalar constants and neighborhood shape.
2. Gene connectivity: The input and output planes of the image processing operator can change.
3. Gene function: The image processing operator itself can change

2.2.5 Discussion

The Genie system has been applied successfully to a wide variety of problems. This is partly due to the co-optimization of feature extraction and classification. Features are found whose output discriminates the true and false classes according to the Fisher criteria. At the same time, the Fisher criteria is more easily met and classification becomes simpler. Another strength of the GENIE system is the combination of well-developed linear and non-linear transformations.

The combination of Genetic Algorithm and Fisher Linear Disciminant is an excellent example of how hybrid optimization techniques can be implemented. It is possible that a Genetic Algorithm will eventually find the Global Optima in any optimization problem. However, the efficiency of search can be substantially improved by incorporating analytical optimization techniques. The EA provides a global sampling of the search space, and then the Fisher Discriminant can quickly find the local optima directly. This increased efficiency is due to assumptions made in the Fisher criteria.

The combination of EA and conventional optimization techniques such as Back Propagation is also seen widely in neural networks [81]. It is also common for EA to be combined with hill-climbing algorithms [82]. These hybrid systems can benefit from both techniques to make search more efficient. The EA provides a robust global search mechanism and hill-climbing or gradient descent methods provide an efficient local search.

Evolutionary Algorithms have been presented as a simple but powerful approach to optimization. If a fitness measure can be calculated for a particular problem, then an EA can be used. The fundamental problem when using EA is computation time. Many candidate solutions must be evaluated, and each evaluation may be complex. This can lead to training times an order of magnitude greater than analytical optimization techniques, and why hybrid optimization techniques are attractive for software implementations. Another way of addressing the large EA computation times, and the approach used in this thesis, is through hardware acceleration with Custom Computers.

2.3 Implementation with Custom Computers

Custom Computers, used in this thesis, are plug-in boards for PC's and workstations. They are intended as application accelerators to which the CPU can pass computationally intensive tasks. Custom Computers usually include a number of reconfigurable logic devices, local memory and a bus interface to a host processor.

Reconfigurable logic in simplest terms is an array of uncommitted gates, or logic functions that can be configured to meet application specific processing requirements. There are many types of reconfigurable logic devices and they are used extensively for rapid prototyping and logic replacement. In this thesis we restrict our attention to FPGAs and particularly SRAM based FPGAs. In Custom Computers the FPGA and host processor are usually tightly coupled. By combining a microprocessor host with FPGAs, an algorithm will benefit from the flexibility of software, as well the performance of specialized hardware.

How well an algorithm can be implemented in CC depends on how well matched the problem architecture is to FPGA devices and the custom computer architecture:

- *The FPGA Architecture:* An understanding of the FPGA architecture is required in order to efficiently implement algorithms on FPGAs. Different FPGAs will be suitable for different problems, depending on the fundamental building blocks that they provide. This is described for FPGAs used in this thesis in Section 2.3.1
- *The Custom Computer Architecture:* The organization of FPGAs, local memory and associated data paths can dictate major design decisions. How the host and CC communicate, and getting data to and from the FPGA must be considered. Further discussion on this topic is presented in Section 2.3.2.

2.3.1 The Building Blocks: Field Programmable Gate Arrays

The FPGA provides flexibility in what function or computation gets performed, as well as how the computational elements are connected. Computation gets performed in Logic Blocks that are distributed on the chip in a particular way. Connections are then made through programmable switch boxes in a routing network. Figure 23 illustrates two organizations of the function units and routing channels.

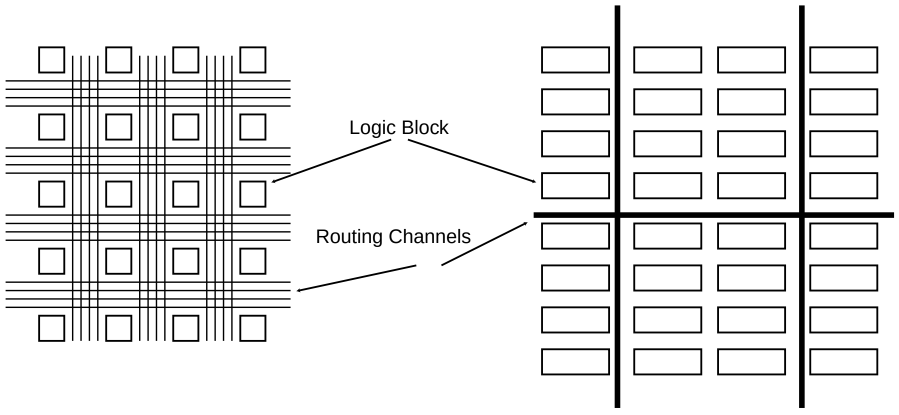


Figure 23: Two example Architectures. Left: Array Right: Channeled

The devices used in this thesis are illustrated on the left and are typical of FPGAs provided by Xilinx [83]. There is a two dimensional array of logic blocks separated by routing channels in both the horizontal and vertical directions. Additional routing and functionality is usually organized hierarchically. Figure 23 also illustrates another organization known as a Channeled architecture. This is an extension of the ASIC standard cell methodology and is typical of FPGAs produced by Altera [84]. In this case, functionality is extended in rows. More predictable routing times are possible with this approach and group of cells are easily placed for multi-bit arithmetic.

An FPGA can be either fine grain or coarse grain. In this thesis, two FPGAs that are representative of these two architectures are described and used. The XC6216 FPGA is a fine grain device and its Logic Block is based around a small multiplexer cell. The Virtex FPGA is a coarse grain architecture and implements much larger, complex logic blocks. In the coarse grain architecture, there are fewer logic blocks per FPGA but each block has greater functionality.

2.3.1.1 *The XC6216 FPGA*

The XC6216 FPGA by Xilinx is depicted in Figure 24 and is described in [85]. It has a regular 64 by 64 two-dimensional array of Logic Blocks. These blocks are very simple, and contain a logic function of 2 variables, a register or flip-flop and routing resources. Cells are grouped hierarchically for routing purposes. At the lowest level cells are interconnected to the nearest neighbors in four directions. These cells are grouped into four and supplemented with length 4 wires. These 4x4 groups then form part of larger 16x16 groups, which are further supplemented by length 16 wires and chip-length interconnects. Higher density devices by Xilinx extend this concept, adding another level to the hierarchy.

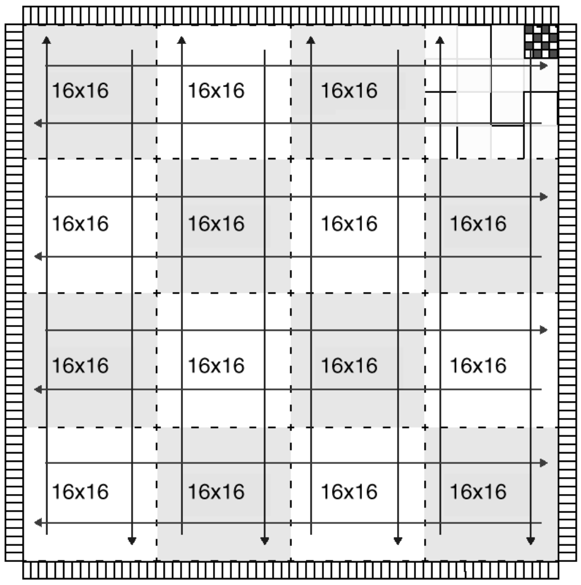


Figure 24: The XC6216 FPGA by Xilinx

The XC6216 is unique to nearly all other FPGAs in how it is programmed. The most common approach to programming SRAM based FPGAs is to download a bit-stream, sequentially to the device. The bit-stream encodes the precise functionality of the *Logic Block* and defines the *Interconnect* between blocks. The XC6216 FPGA implements this differently. The configuration registers used to program the XC6216 are memory mapped to the host processor. This essentially provides the host with random access to the FPGA configuration. This is known as Rapid Reconfigurability and is particularly attractive in situations where only a small part or the design needs to be reconfigured. This is called Partial Reconfiguration and is discussed further in Chapter 3.

Another important aspect of the XC6000 series FPGAs that has not been present in other devices is protection from internal contention. By using multiplexers on every input node Xilinx has ensured that no combination of configuration bits can cause multiple sources to drive internal wires and damage the device. In previous devices it was the responsibility of the designer and CAD software to ensure that this did not occur.

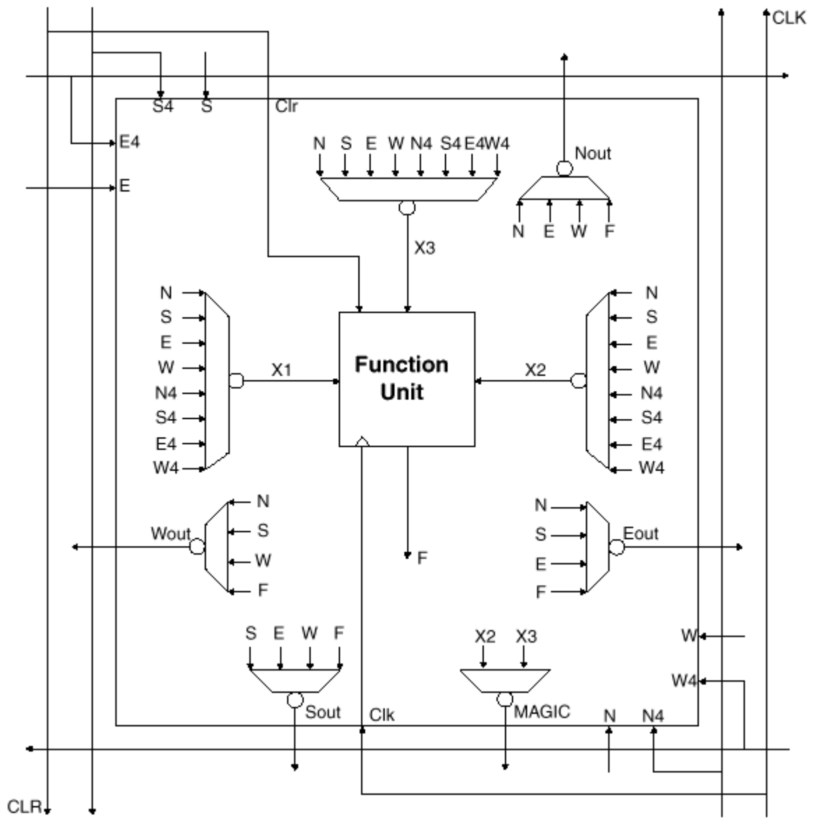


Figure 25: XC6216 Logic Block

The Logic Block of the XC6216, is depicted in Figure 25. It has 24 bits of configuration memory. These configure the cell's function unit as well as the local routing resources. The function block is capable of implementing any 2-input logic function or 3-input multiplexer. It also has a D-type flip-flop. Each cell receives input from its four closest neighbors and several other more distant cells. For example, in Figure 25: N, S, E and W represent nearest neighbor communication and N4, S4, E4 and W4 represent interconnection between neighboring 4x4 blocks. Additional configuration bits define global routing resources and the functionality of I/O blocks found on the perimeter of the array.

Both the fine grain nature of the XC6216 FPGA, and its hierarchical routing suggest the device is most suitable for logic-based computations. This is described more in Chapter 4. When more complex processing is required, such as multi-bit arithmetic, the fine grain architecture can be inefficient. For this reason, many modern FPGAs such as the Virtex FPGA from Xilinx have moved to coarse grain architectures that are more suitable to computation at the arithmetic level.

2.3.1.2 The Virtex FPGA

The Virtex FPGA by Xilinx is the other FPGA used in this thesis. It was first released in 1999. The largest FPGA available in this series is currently the XCV3200E FPGA, which uses a 0.18 um 6-layer metal process and has approximately 4 Million system gates. In this thesis the XCV2000E FPGA is used which is approximately 0.6 times the size. Figure 26 illustrates how the Virtex Logic Blocks (CLBs) are organized. An additional component, seen in Figure 26 is dedicated on-chip memory, known as Block Rams (BRAMs). Computation usually requires memory and a key advantage to FPGA implementations is an ability to design specialized memory interfaces to fast, high bandwidth memory. The importance of Block RAMS in this thesis becomes clear in Chapter 6 with the implementation of spatial convolutions.

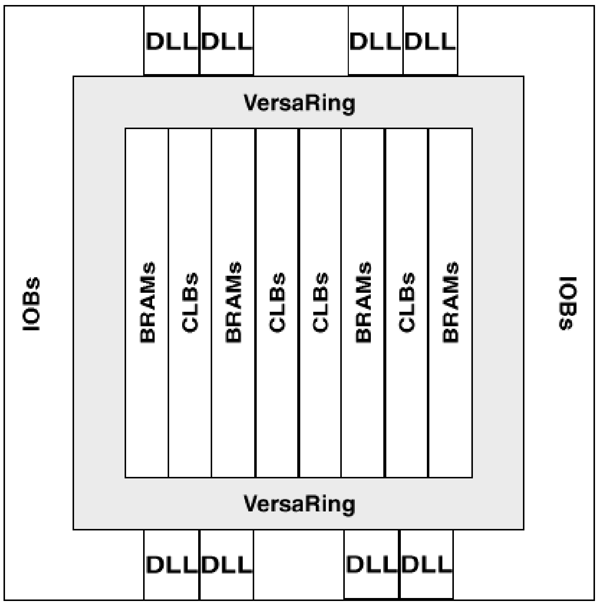


Figure 26: The Virtex FPGA by Xilinx

Figure 27 depicts the Virtex Logic Block. It is made up of two *slices*. Each slice has two 4-input Look Up Tables, and 2 registers. In addition, Logic Blocks contain specialized resources for carry propagation logic seen commonly in multi-bit arithmetic. Each slice can implement a 2-bit adder and when Logic Blocks are stacked, flexible bit-width arithmetic with low propagation delays can be implemented.

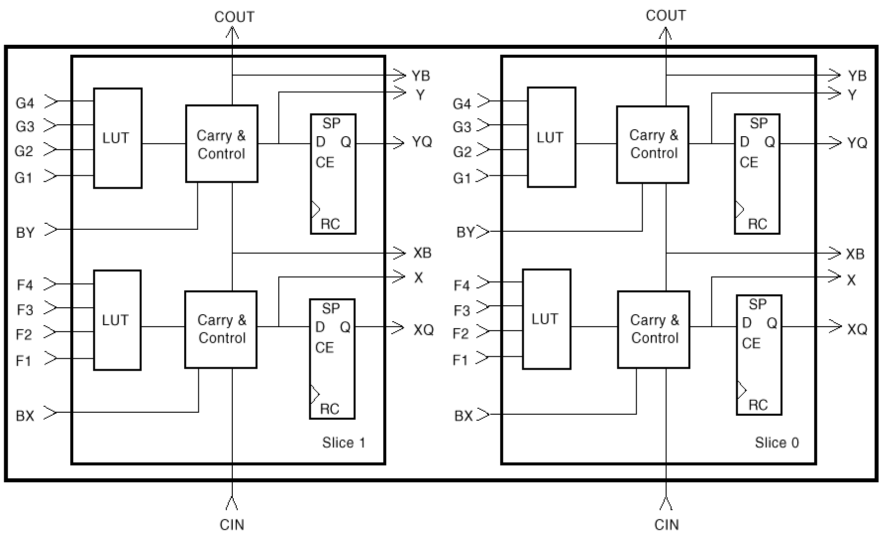


Figure 27: Virtex Logic Block

Also provided in the Carry Logic is a dedicated AND gate which can be used to efficiently implement integer multiplication. The number of Logic Blocks required to implement array multipliers is effectively halved. The Virtex FPGA architecture suggests fine grain parallelism using simple processing elements similar to the XC6216. However, the more complex Logic Block of the Virtex means processing elements can include multi-bit arithmetic, small multipliers and on-chip registers. This is described more in Chapter 5.

The Virtex FPGA also supports partial reconfiguration [86]. However, it is more limited than in the XC6216 FPGA and the device is not considered Rapidly Reconfigurable. Each column of Logic Blocks in the Virtex FPGA can be independently addressed in frames. There are 48 frames associated with each column and 80 rows of Logic Blocks per column in the XCV2000E FPGA. To reconfigure a particular Logic Block requires multiple frames since the information is stored column wise. Another limitation of rapid reconfiguration in Virtex FPGAs is the fact that Xilinx has commercial interest to not publish the configuration bit-stream format publicly. Without this information, it is impossible to dynamically reconfigure the device at the bit-stream level. Xilinx introduced a Java based tool known as JBITS as an alternative. This program mediates bit-stream modifications, allowing the user to specify logic block and routing configurations, which are then mapped to the proprietary bit-stream.

Use of the Virtex partial reconfiguration for implementing EA experiments is yet to be seen. This is partly due to the frame organization, but mostly due to lack of tools. Also, many Custom Computers that use Virtex FPGAs do not provide support for this feature at the board level. This is the case for the Firebird CC used in this thesis and therefore the Virtex partial reconfiguration capabilities were not explored.

2.3.1.3 Programming FPGAs

While the design cycle for an FPGA can be significantly lower than for an ASIC, it is still much longer than software design cycles. The design process for a typical FPGA application is illustrated in Figure 28. The design is first specified using either graphical schematic entry or Hardware Description Languages (HDLs). HDLs are comparable to software programming languages, but their functions model hardware constructs such as gates and flip-flops. There are many varieties of HDLs that have been developed, usually for specific target architectures. Two of the most commonly used are Very-high-speed-integrated-circuit Hardware Description Language (VHDL) [87] and Verilog.

Once specified, the design is then simulated using software tools to verify its accuracy. Specification and Simulation is usually an iterative process that can take days or weeks depending on the design complexity. Once a design is verified through simulation it then must be compiled for a particular FPGA. This process is carried out in Stages 3 through 6 and is largely automated using software tools. In Synthesis, the design is converted to an intermediate format, known as a netlist, which is device independent. Technology Mapping then translates the device independent netlist to a device specific format.

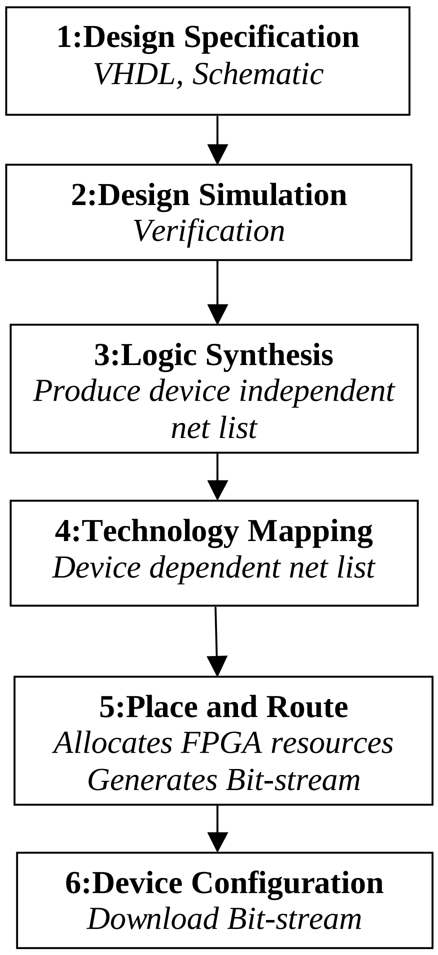


Figure 28: Typical FPGA Design Process

The 5th step in the design process is Place and Route, which is the most time-consuming of the automated tasks. It involves physically allocating the functional units on the FPGA and then routing the required interconnections. This process is similar to that used for ASIC design and is important in terms of efficiency due to the limited logic and routing resources available in FPGAs. Automated design software does not always perform well, and it is common for a designer to manually place some components during this step to help the routing software. The final step in the design process is to generate the configuration bit stream used to program the FPGA device. This is a simple translation and usually takes a few seconds. These configuration bits can then downloaded into the FPGA device in milliseconds.

2.3.1.4 Discussion

Many FPGAs manufacturers appear to be moving towards coarser grain, higher complexity architectures with specialized arithmetic and logical components. The future of FPGAs may lie in devices that incorporate both microprocessor and FPGAs on the same chip. Already, both Xilinx and Altera offer soft processor cores that can be used with their devices. Xilinx has produced an optimized 32-bit processor for their soon to be released Virtex-II part which is called the MicroBlaze [88]. Xilinx also announced plans in November 2000 for a true hybrid system called Xilinx Empower that will incorporate PowerPC 405 microprocessors from IBM with Virtex-II reconfigurable logic architecture [89]. It is likely that devices with RISC microprocessor based Logic Blocks are not too distant!

Future FPGA architectures will no doubt have significant effect upon the direction of work that follows this thesis. Algorithms that need a combination of fixed-bit data intensive image processing and more complex arithmetic such as matrix inversions could potentially benefit. The combination of EA and conventional classifier seen in GENIE is an example of such an algorithm.

At the deepest level it is the FPGA architecture that dictates what algorithms will be efficiently implemented. This is particularly true for modern FPGAs capable of complete system-on-the-chip implementations. However, at a higher level there are other considerations: the way FPGA devices are connected, implemented and used with respect to a host processor.

2.3.2 Custom Computers

The most prevalent type of Custom Computer is a plug in device that communicates with a microprocessor through an expansion bus. Custom Computers have been developed and implemented by many research institutions and commercial companies. There are many architectural choices that must be made in building a Custom Computer. These often depend on the type of problems the machine is intended for. These choices include memory organization, data-path widths and device interconnection. These issues are explained through examples of custom computer that have been built and used in the research community.

Device interconnection was of particular importance in the late 1980's and early 1990's when Custom Computers were first developed. This was due to the large number of FPGA devices that had to be used to provide sufficient resources for practical problems. Modern high density FPGAs have helped a lot with this problem. One of the first boards to be developed was the DEC PeRle-0, produced in the DEC Paris Research Laboratory in 1989 [90]. This board implemented 25 Xilinx 3020 FPGAs in a 5 by 5 array. A similar sized board, known as Splash was also built by the Supercomputing Research Center in 1989.

2.3.2.1 An Early Custom Computer

Splash-2 is an extension of the original Splash board, and was also developed at the Supercomputing Research Center, Maryland [91]. The Splash-2 architecture is illustrated in Figure 29 and consists of a Sun Sparc Station host, an interface board and one or more Spash-2 array boards.

Each Splash array board implements 17 Xilinx XC4010 FPGAs. 16 of these are connected in a linear array, which is ideal for systolic or pipelined processing. Additionally, 512 Kbytes of memory is attached to each FPGA. This is an excellent example of how memory is usually implemented on CC boards and is known as Local Memory. In the Splash-2 and most custom computers, this is necessary to avoid problems with the limited memory bandwidth through the global bus.

FPGAs provide the computational resources to implement a particular algorithm. While this is sufficient for highly systolic computations, most algorithms require more flexible memory access. Applications can be accelerated with CC by using specialized memory address hardware. A good design means that all parts of the computation have access to their stored variables, as they are required. Distributed memory, where there are multiple, smaller memories distributed on a board, provide large bandwidth to stored variables. This means designs can be implemented efficiently and algorithms accelerated. Modern FPGA devices are also designed with this in mind, and now have sufficient input/output resources to enable 1 FPGA to access 5 or more local memories. This will be described in more detail with respect to the Firebird CC.

The interface board in the Splash system contains a programmable clock and provides DMA access between the host and local memory. Two user programmable FPGA's (XL and XR) can be configured for application specific data stream pre-processing and post-processing as data is supplied to and fetched from the Splash array boards. A variety of both synchronous and asynchronous communication paths exist between the Host and the array boards. These are used to supply and fetch data at run time and configure the FPGA surface.

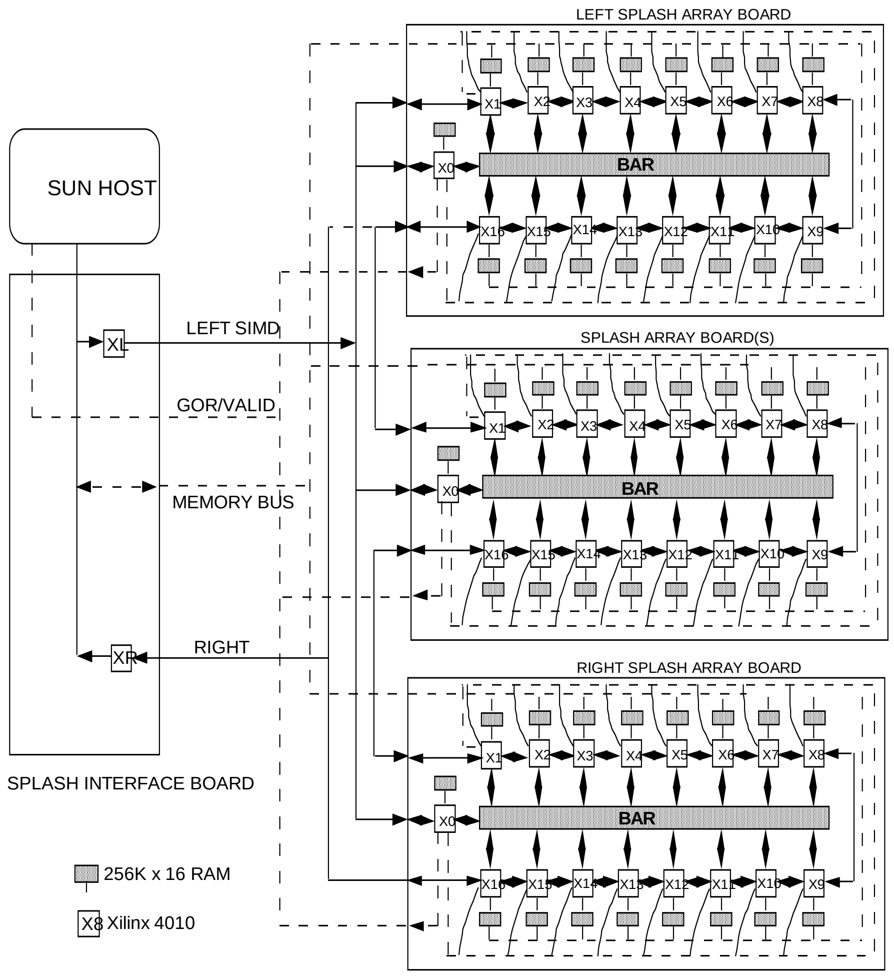


Figure 29: Architecture of Splash-2 System [91]

The Splash-2 was applied to a number of problems including searching genetic databases, fingerprint matching and Automatic Target Detection (ATD). Due to the limited resources available on each XC4020 FPGA, implementation on the Splash-2 requires problem partitioning. This means the problem is decomposed into a number of components, each of which is implemented on a different FPGA. Streamed or pipelined algorithms are most easily decomposed with the Splash linear array architecture [91].

2.3.2.2 *Custom Computers for Real-Time Image Processing*

A major consideration in producing a Custom Computer capable of real-time processing at the sensor is data IO. It is important that data can be sent to and received from the CC at high speed. It is also important that CC data paths between FPGAs and local memory do not limit data throughput. The Giga-Ops Corporations G800 custom computer was designed specifically for high-speed digital signal and video processing [92], and therefore of interest to this thesis.

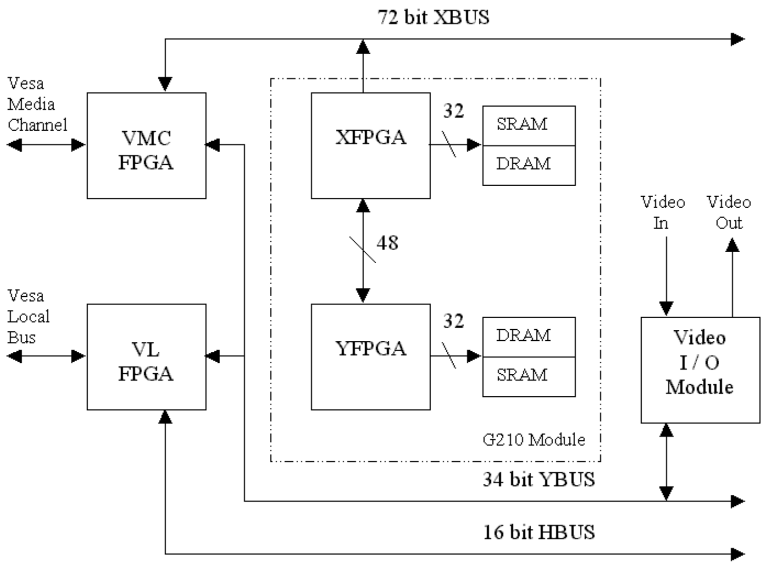


Figure 30: Architecture of G800 Custom Computing Board with G210 Module

The G800 itself is a baseboard, which is then extended as required to meet application specific requirements. The G800 is illustrated in Figure 30. The control logic is implemented in two XC4003H FPGAs from Xilinx. They implement interfaces to a Vesa Local Bus as well as a Vesa Media Channel Bus. Up to 16 G210 computing modules, containing FPGAs and Local Memory can be installed on the baseboard. The G210 is also illustrated in Figure 30 and contains 2 XC4010 FPGAs from Xilinx, as well as a 32 bit memory port for each FPGA. In addition, one custom input / output module such as a video frame grabber / encoder can be installed.

The G800 system is particularly well suited for real time image and video processing applications. This is due to the dedicated video buses, and availability of standard IO cards for translating composite video to NTSC.

The RCA-3 is another Custom Computer architecture that has flexible, high-speed IO and is intended for real-time processing. It was developed relatively recently at the Los Alamos National Laboratory and is illustrated in Figure 31.

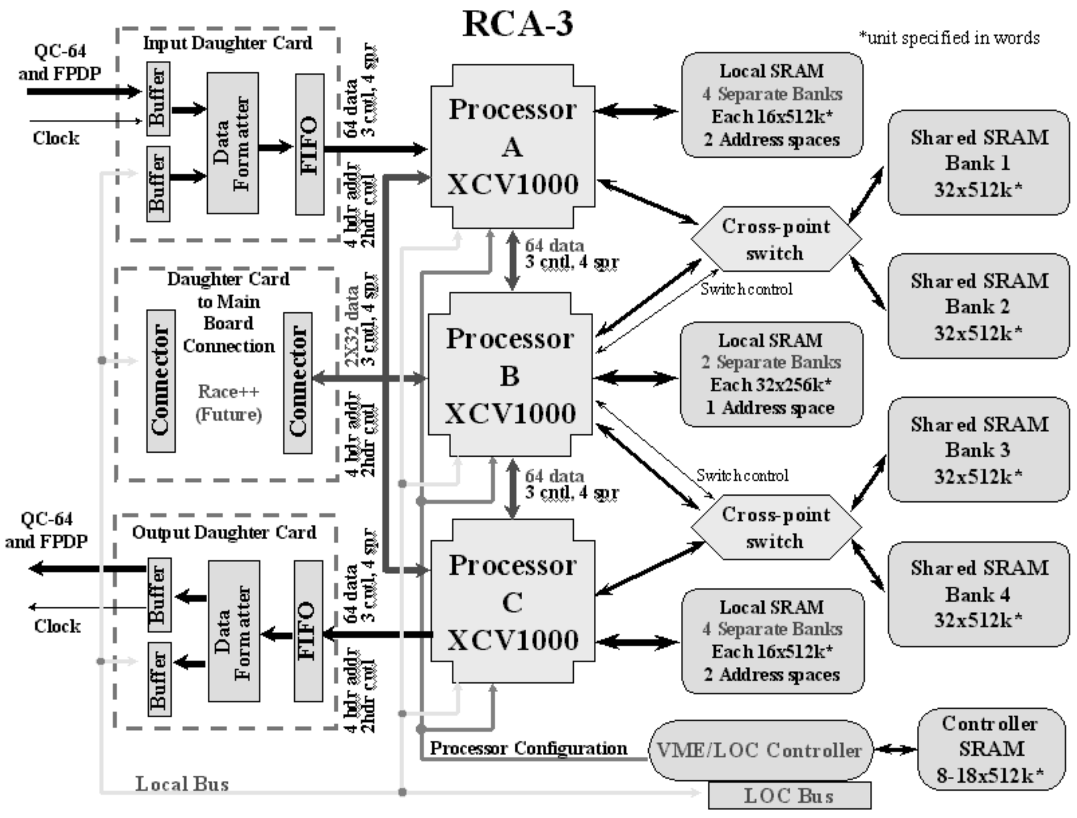


Figure 31: The RCA-3 Architecture

The RCA-3 is based around 3 FPGA processors. The fewer number of FPGAs is indicative of the increased capacity of modern FPGA devices. Processors A and C each have 4 independent banks of memory. Additionally, each Processor has access to a shared memory bank indicated on the right of Figure 30. This is to provide large block transfer capability between pairs of processors. A and B processors are paired, as well as processors B and C. This is an excellent design choice and allows the memory to be toggled between processors. For example, processor A performs some computation on the input stream and writes its result to shared memory bank 1. Shared memories 1 and 2 are then toggled. Processor B can begin processing the new block in shared memory 1 while Processor A prepares the next block in shared memory 2. Memories can be toggled in one clock cycle, which is ideal for pipeline processing of block-orientated data [93].

The RCA-3 was designed for high-speed IO and has room for 3 I/O daughter cards. LANL has developed daughter cards that implement the Front Panel Data Port (FPDP) standard as well as a VXI bus interface. These daughter cards provide First-In-First-Out (FIFO) memory, which is ideal for using the RCA-3 in a larger, often asynchronous system.

2.3.2.3 Custom Computers in this thesis

Two custom computers are used in this thesis. One is based around the Xilinx XC6216 FPGA that was described in Section 2.3.1.1, and is called the HotWorks CC built by Virtual Computer Corporation. It has a single XC6216 FPGA and communicates through a 32-bit PCI bus to standard host computer. Implementations in this thesis did not use the local memory available and therefore further discussion is left to Chapter 4.

The second custom computer, used in the subsequent chapters, is the Firebird CC from Annapolis Microsystems. The Firebird is based around a single Virtex 2000E FPGA from Xilinx. This device was described in Section 2.3.1.2 and has substantial computational resources and large quantities of distributed on-chip RAM. Since only 1 FPGA is used, the problem of partitioning encountered in early CC implementations is avoided.

Using only one FPGA also makes the custom computer architecture very simple. Figure 32 illustrates the Firebird CC. There are five independent banks of local memory: four 64-bit memory banks, and a 5th 32-bit memory. In total, this means there is a 288-bit data-path between the FPGA and the memory. On the left of Figure 8, a 64-bit 66MHz PCI bus connects the FPGA to the host computer. Additionally, a dedicated I/O port that could potentially be used for real-time processing is seen on the right of Figure 32.

The Firebird has significant bandwidth to local memory and a standard PCI interface, and therefore the custom computer architecture did not significantly affect design choices in this thesis. The Firebird CC does not support the partial reconfiguration available in the Virtex FPGA, and the entire device must be configured at once. Alternatives to partial reconfiguration are described in Chapter 3.

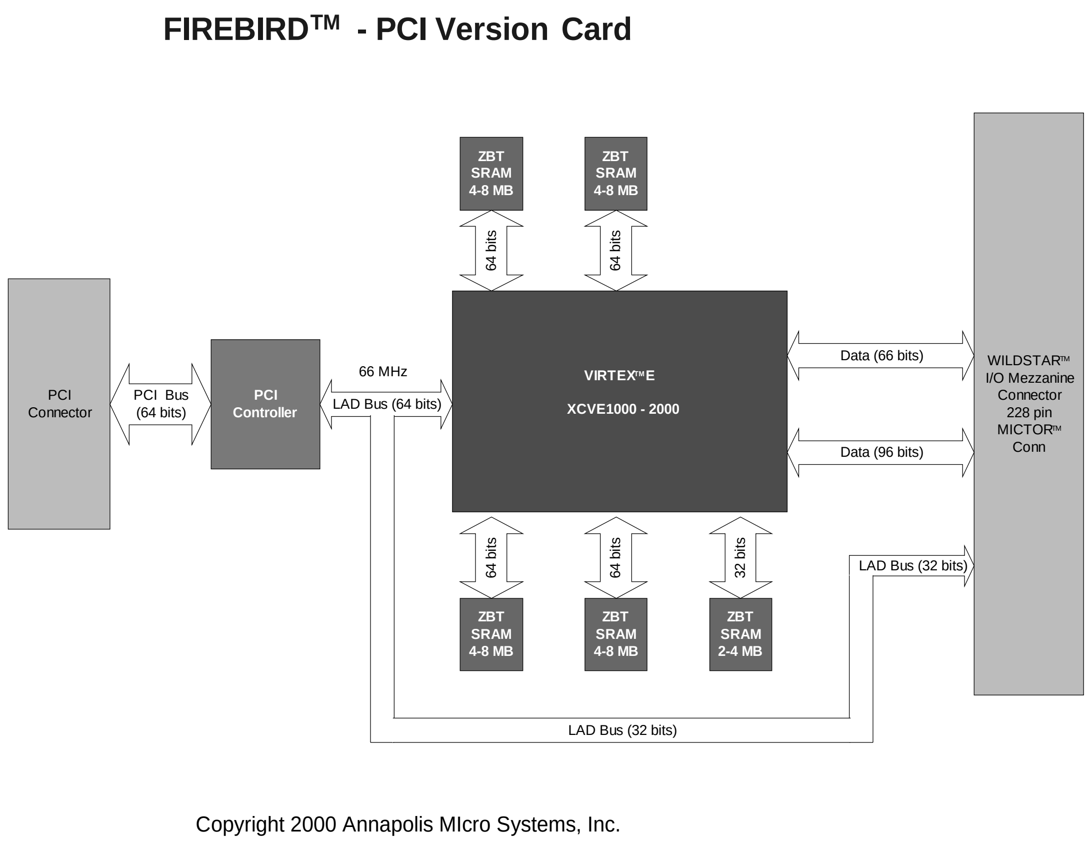


Figure 32: The Firebird RCC

2.3.3 Evolvable Hardware

The building blocks for *hardware efficient* AFE have now been described. These come from two sources: algorithmic components of conventional AFE solutions, as well as the hardware resources available in FPGA devices. Background to EA, and how they might be used to optimize these building blocks was also described. Before continuing, it is worth considering how EA have been used in combination with FPGA devices in literature.

The combination of Reconfigurable Logic devices and EA optimization in the field of Evolvable Hardware (EHW) was suggested independently in Japan and Switzerland in 1992 [94]. Today, there are several international conferences and workshops, dedicated journals and also Evolvable Hardware sessions in most major FPGA and Reconfigurable Computing meetings.

2.3.3.1 Classification of EHW Approaches

In this thesis, the term Evolvable Hardware is used to refer to a broad range of approaches with the common components of EA and FPGAs. However, two classes of experiment can be considered. The first is motivated by the long execution times of EA experiments in software. Such experiments have application specific implementation requirements and the flexibility of CC makes it a natural choice for accelerating EA experiments. This is a primary motivation of this thesis.

The second class of experiment is concerned with hardware design itself, and therefore the EA is usually applied to more primitive hardware building blocks. Over the last few years EHW has even grown to include using EAs for antenna design, vehicle design and other physical design problems. However, the distinction between these two approaches is often blurred. In the first case, accelerating software EA experiments often requires algorithmic trade-offs in order to implement chromosomes efficiently on FPGAs. In the second case, knowledge of high-level algorithms is often required in order to apply EHW design principles to practical problems.

A classification of EHW that is better defined concerns where the EA fitness evaluation takes place. *Intrinsic* EHW is when the hardware device is used for the fitness evaluation. The approach of this thesis can be considered *intrinsic*. There is a large body of work on *extrinsic* EHW. In this approach software models or analogue circuit simulators are used to evaluate fitness. Further discussion is restricted to *intrinsic* EHW since Custom Computer acceleration of EA experiments is a primary motivation.

2.3.3.2 EHW Devices

FPGAs are a popular choice in EHW due to their flexibility and speed. A great deal of work in EHW has used the XC6200 series FPGAs. This is mainly due to Rapid Reconfiguration, which means chromosomes can be swapped in and out of the FPGA very quickly [94]. However, they must not be considered the only devices capable of implementing evolvable hardware techniques. It has been suggested that EHW be applied to memory-based devices directly [95]. Dedicated architecture devices would also be suitable for evolutionary design approaches such as neural network and fuzzy logic chips.

Kajitani et al. in [96] built a custom EHW device that included reconfigurable logic, large on-chip memory and a 16-bit CPU. This device was used to evolve a prosthetic hand controller. The EA was accelerated by a factor of 62 compared to the same GA implemented on a 200MHz Sparc2 workstation. The same research group in [97] also built and implemented the GRD Chip, which includes a 16-bit CPU and a binary tree of 16 Digital Signal Processors (DSP). A genetic algorithm is implemented on the CPU, which optimizes a neural network implemented on the 16 DSPs. Nine GRD chips were used to evolve an adaptive equalizer for digital communications, and achieved a speed-up of 160 compared to a Sun Ultra2 200MHz workstation.

2.3.3.3 FPGAs as Soft Silicon

Work by Thompson in [98], [95] is a good example of the potential of *intrinsic* evolution. In this work an EA is applied directly to the FPGA configuration bit-string. The device used was the XC6216 rapidly reconfigurable FPGA. The experiment aimed to optimize a circuit that would discriminate between 1kHz and 10kHz square waves. A designated output pin should be high for one frequency and low for the other. The fitness evaluation was measured by directly observing the designated output pin. No constraints were placed on the XC6216 configuration bit-string, and binary mutation and crossover could be applied directly.

Thompson's non-constrained experiment produced several configurations that took advantage of the continuous-time dynamics of the FPGA device. These analogue solutions proved deceptively difficult to analyze but produced surprisingly accurate results. The solutions produced were found to rely on the detailed physics of the hardware: time delays, parasitic capacitances, cross-talk and other low level characteristics of the particular FPGA.

The circuits evolved with Thompson's approach are inherent to the type of FPGA used. This has led some researchers to ask the question: could a different type of Reconfigurable Logic exploit this technique more effectively? Work in [99] and [100] describe some of the design choices and evolution at the transistor level has been suggested in [101].

2.3.3.4 Accelerating EA Experiments

The Splash-2 was one of the first custom computers to be used to accelerate EA experiments [102]. Solutions to the Traveling Salesman problem were found by using 4 of the 16 FPGAs and achieved a speed-up of 10.6 compared to a 125 MHz workstation. About the same time, another custom computer called the Armstrong III was used in [103] to solve a circuit-partitioning problem. Speed-up of 8.6 was observed compared to a 60Mhz workstation.

In the experiments just described, the entire GA algorithm, including fitness evaluation and genetic operators were implemented in the custom computer. The GA is inherently parallel, and simple in structure and therefore can be greatly accelerated in this way. Other examples of this approach include [104] where a steady-state GA was implemented on a Lucent Technologies FPGA and achieved speed-up of 730 compared to a 333MHz DEC Alpha. Recently in [105] a Virtex XCV300 FPGA was used to achieve speed-up of 320 compared to a 366MHz Pentium II for a protein-folding problem. In this work, authors project that by using the larger XCV3200E FPGA that speed-up of 9600 could be achieved.

Several other researchers have suggested complete GA-pipelines for FPGAs. Tufte and Haddow [106] presented a GA pipeline that is targeted at Virtex FPGAs but is yet to be implemented. Sidhu et al. in [107] describes a Genetic Programming pipeline implemented on a XC6264 FPGA. Speed-up of 19 compared to a 200MHz Pentium Pro was obtained for an arithmetic regression problem, and three orders of magnitude for a logic-based multiplexer problem.

Implementing the entire GA on custom computers has been shown to achieve significant speed-up compared to software implementations. However, since the host and custom computer are usually tightly coupled, co-processor architectures are also possible. In [108] a GA co-processor using a Virtex 1000 device was used to solve an Iterated Prisoner's Dilemma. Speed-up of 200 compared to a 750MHz Pentium-III was reported. Authors also suggest that by using the partial reconfiguration and read-back functions of the Virtex FPGA, speed-up of 400 could be expected. Koza in [94] used a co-processor approach with the XC6216 FPGA. For reasons that are described in Chapter 3, this thesis also uses a co-processor approach.

2.3.3.5 EHW applied to AFE

One of the earliest EHW efforts to address problems related to AFE, was carried out in the Electro-Technical Laboratory in Japan [109]. In this work EA was applied to Programmable Logic Devices (PLD). They used a variable-length encoding scheme that dictated specific connections in the PLD AND-OR array. Their system was applied to a binary character recognition problem. A good account of the subsequent work by this group can be found in [110]. Much of this work is based on custom EHW chips that were described in Section 2.3.3.2. More recently in [111], the XC6264 FPGA was used to evolve a 9by9 linear spatial filter.

Use of EA and FPGAs for practical AFE problems is yet to mature. In fact, at the time of writing, very few implementations have been reported that reach the level of complexity that this thesis develops. This is partly due to the fact many EHW researchers are interested in applying EA to low-level building blocks. Thompson's work [98] is an excellent example of the potential of this approach. Before continuing, two other areas relevant to this thesis are briefly described. These are FPGA implementations of evolutionary neural networks and FPGA implementations of AFE algorithms that do not use EA.

Just prior to submission of this thesis, work in [112] presented a novel block based neural network. This network is more easily implemented in FPGAs than traditional neural networks and is optimized using EA to solve a robot controller problem. However, the network has not been implemented. Another interesting evolutionary neural network that has been implemented in FPGAs is reported in [113]. The custom computer is called the CAM-Brain machine and uses 72 XC6264 FPGAs. A neural network variant is implemented within a cellular automata model, and an EA is used to optimize the connectivity and weights of the network to solve a robot controller problem.

Work by Figueiredo et al. [114] describes implementation of a probabilistic neural network for multi-spectral image classification. The network was applied to the spectral dimension of the raw image cube without feature extraction and spatial information was not included in the classification. Other Neural Network implementations are described in Chapter 5.

2.4 Chapter Summary

The Literature Review has described all the basic elements from which this thesis builds. In doing so the design space for hardware AFE has been defined. The major topics relevant to future chapters are:

- Extracting useful information from images where noise is hard to characterize requires both linear and non-linear filters.
- Feature enhancement/extraction and classification are important and closely tied aspects of AFE problems.
- Evolutionary Algorithms are a computationally intensive, but flexible, and can be applied to models that are traditionally hard to optimize.
- Custom Computers have flexible resources to implement application specific hardware and can provide substantial speed-up compared to software implementations.
- Examples in Evolvable Hardware demonstrate how EA can be applied to low-level FPGA building blocks to find extremely hardware efficient solutions to various problems.

A common theme of this thesis is *hybrid* architectures. Examples of this, identified in the Literature Review are hybrid linear / non-linear filters [Section 2.1.1], hybrid feature extraction / classification architectures [Section 2.1.4], hybrid optimization schemes [Section 2.2.5], and hybrid Microprocessor / FPGA devices [Section 2.3.1]. Hybrid approaches are attractive since they often can combine the best parts of multiple techniques. While the relationship between this and crossover is probably chance, it is safe to say that the flexibility of EA means hybrid approaches can be optimized as easily as their constituent parts.

In the context of design spaces discussed in the Introduction, the approach of this thesis is a hybrid of FPGA based EHW and software AFE algorithms. Design spaces to be described are dictated largely by the combination of computational resources available in FPGAs and algorithmic components of traditional AFE.

Chapter 3

Design Considerations

This chapter is intended as a design guide for readers interested in accelerating EA experiments using custom computers. The major design choices are described as well as more detailed descriptions of hardware architectures used in the thesis. This chapter is not concerned with AFE particularly although image processing architectures are described. Readers interested more in experimental work can begin with Chapter 4.

Maintaining a population of solutions means EA can produce robust solutions in many problem domains but it also leads to large computation times. Evolutionary Algorithm (EA) experiments can be implemented on Custom Computers (CC), with substantial speed-up compared to software implementations. To obtain significant speed-up there are three main design choices that must be considered. These will be described in turn:

1. How the experiment is divided between the host processor and CC.
2. How the chromosome will be implemented using CC.
3. How chromosomes will be swapped in and out of the CC.

3.1 Division of Labor

A major design choice when implementing EA experiments is the division of labor between host and CC. By tightly coupling the host and CC both the flexibility of software and the performance of application specific hardware can be exploited. In tightly coupled systems, one of the main design considerations is the communication between the host and CC. Communication across the global bus is generally much slower than CC computation and therefore must be considered carefully.

3.1.1 The Fitness Evaluator

Usually the most computationally intensive part of EA is the fitness evaluation. This involves evaluating how well each individual in the EA population solves the particular problem. Fitness evaluation is composed of:

1. Chromosome Execution: A candidate solution is applied to the training data.
2. Fitness Metric: A measure is derived based on the success of the chromosome execution in achieving the desired result.

When an EA are applied to image processing problems, chromosome execution can involve several data intensive operations. These data intensive operations can be greatly accelerated by the CC. To minimize communication from CC-to-host, it is desirable to also implement the Fitness Metric in the CC. This way, the host program only needs to retrieve the fitness score for each evaluation.

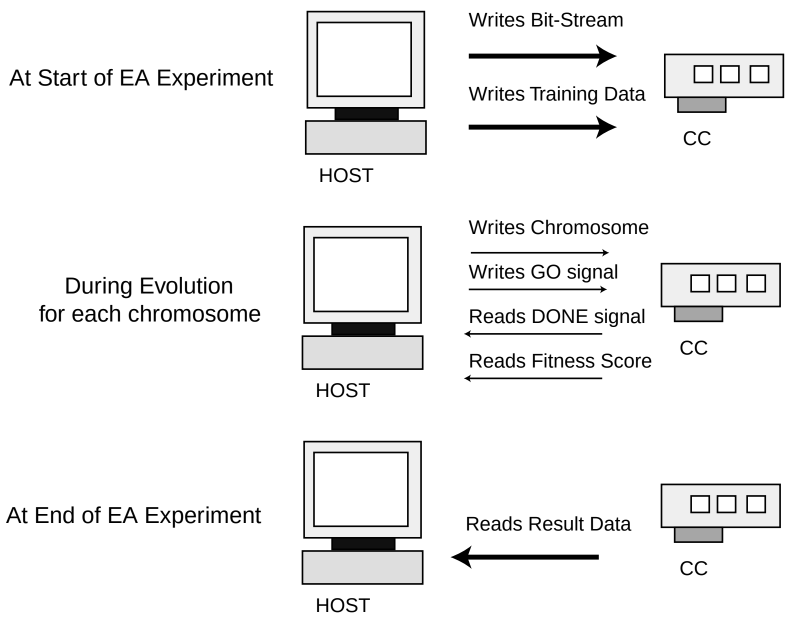


Figure 33: Communications for Fitness Evaluator Architecture

The host/CC communications for the Fitness Evaluator is illustrated in Figure 33. Large volume training images are loaded once at the start of an EA run, to the CC local memory. Communication between host and the CC during the evolution involves downloading a particular chromosome, initiating the chromosome evaluation, and then retrieving the fitness score. Only at the end of the EA run, is the result of chromosome execution retrieved for inspection.

Sometimes it is convenient to calculate the Fitness Metric in software. Reasons for this include:

1. It is difficult to calculate in CC due to the chromosome architecture. This is the case for experiments to be discussed in Chapter 4.
2. The Fitness Metric is complicated and may require complex hardware such as floating point arithmetic.
3. The best Fitness Metric is not known and increased flexibility is required.

When the Fitness Metric is calculated in software, the CC-to-host communication is greater since the result of chromosome execution must be returned to the host for each chromosome. In the case of image processing, the result is often an entire image, and therefore this communication can be large.

All implementations in this thesis use the single chromosome fitness evaluator architecture described. This is due to complexity of problems addressed, which leads to complex chromosomes that push resource limits of state-of-the-art FPGA devices.

Many of the EA implementations that were described in the Literature Review use the single fitness evaluator architecture, but also implemented the EA in hardware. The significant speed-up achieved by some of these implementations was reported. A problem with this approach is lack of flexibility. Due to the experimental nature of this thesis the co-processor approach was used so that representation and EA variants could be explored. Also, the computation times for AFE fitness evaluation in this thesis is large compared to the GA algorithm itself and therefore significant speed-up can be obtained with the co-processor approach. It is foreseeable, that the EA could be moved to the CC, once this research matures, to obtain greater performance.

Before continuing, it is worth considering divisions of labor when more than one fitness evaluator is implemented in the CC. In this case, the motivation for moving the EA to hardware is stronger.

3.1.2 Multiple Fitness Evaluators

A spectrum of host/CC architectures can be imagined, which is depicted in Figure 34. On the left of Figure 34 is the Fitness Evaluator architecture. In this model the CC implements one chromosome at a time.

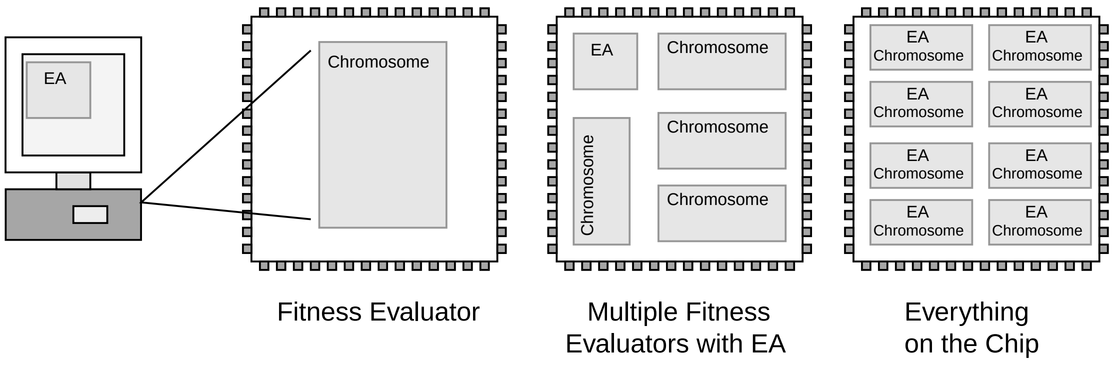


Figure 34: The Division of Labor

The EA uses a population of chromosomes in optimization and therefore is inherently parallel. Multiple fitness evaluators can therefore be implemented to achieve speed-up. At the limit of this approach an entire EA population could be implemented in CC. However, since each chromosome must be configured and its fitness score retrieved by the host processor, communication between the host and CC can become large. A solution to this problem is to move the EA itself to the CC. This is the 2nd model illustrated in Figure 34 and can lead to speed-up in 3 ways:

1. Speed-up through CC acceleration of chromosome evaluations.
2. Speed-up by evaluating many chromosomes in parallel.
3. Speed-up by avoiding relatively slow host/CC communication.

The 3rd architecture in Figure 34 takes this a step further. In a massively parallel system (if enough chromosomes can be implemented) it is often desirable to avoid complex centralized control. A novel solution to this problem is to distribute the EA amongst the multiple Fitness Evaluators. How the EA is applied in this situation is an interesting topic, which is described in detail in a paper co-authored with Simon Perkins, by the author: *Everything on the Chip* [115]. This paper is included for reference in Appendix B. In this work an entire population of non-linear *Stack Filters,* which are described and used in Chapter 4, are evolved completely in CC.

A problem with moving the EA to the CC is loss of flexibility. Representation is generally restricted to bit strings, and genetic operators must be kept simple. Such restrictions may be appropriate for some problems. In *Everything on the Chip* [115], these restrictions dictated evolution of a novel superset of Stack Filters without a loss in performance. As FPGA devices become larger, it is believed multiple fitness evaluator architectures will become more useful for practical problems.

3.2 Chromosome Architectures

The second design consideration concerns the chromosome architecture. EA experiments will benefit most from CC implementation if the chromosomes can be mapped efficiently to the FPGA device. Exactly what algorithms can achieve significant speed-up on FPGAs is still a question of research. Image, video and signal processing has received particular attention since they are data intensive [116]. Some ways by which FPGAs gain significant speed-up include:

- Algorithmic Parallelism: The algorithm has inherent parallelism. This is often true in image processing algorithms.

- Data Intensive: This is typical of image processing algorithms, which use simple computational elements but are required to process very large amounts of data. These types of algorithms usually perform poorly on microprocessors due to bottlenecks in the data flow. Custom computers can take advantage of the algorithm specific data requirements and implement high throughput pipelined structures.

- Tailored Data-paths: Since the FPGA surface operates at the bit level, data-path bit-widths can be tailored to the algorithm. This can lead to significant resource savings and therefore more efficient use of hardware.

- Tailored Arithmetic Units: In an FPGA, arithmetic operations can utilize on-chip memory and instruction-level pipelining. This reduces the number of memory accesses required to complete a sequence of operations, which would normally dominate computation time in a conventional microprocessor. Note, FPGAs can implement fixed-point arithmetic efficiently, but currently lack computational resources for efficient floating-point arithmetic. It is possible future FPGA devices will include dedicated floating-point units, which may help with this problem.

- Flexible Address Generation: The specific memory and address generation requirements of an algorithm can be optimized to minimize contention. This means all components of an algorithm can have access to variables, as they are required.

Another important consideration in the hardware design of large, complex algorithms is modularity. A modular approach attempts to decompose the problem into a number of components, each of which is simpler to implement than the complete algorithm. Modularity can be taken a step further in which the algorithm is decomposed into a number of identical components. This is particularly attractive due to the homogenous nature of FPGA architectures. In this case, components are often referred to as *cells*. This can simplify hardware design significantly and is an approach often used in this thesis.

There are two main ways of achieving speedup in image processing within the context of a modular architecture. These depend on how cells are connected on the FPGA, which leads to either pixel parallelism or instruction-level parallelism. The two extremes of this approach are illustrated in Figure 35, but any combination of pixel and instruction-level parallelism is possible. The next two sections will describe pixel parallelism and instruction-level parallelism in more detail.

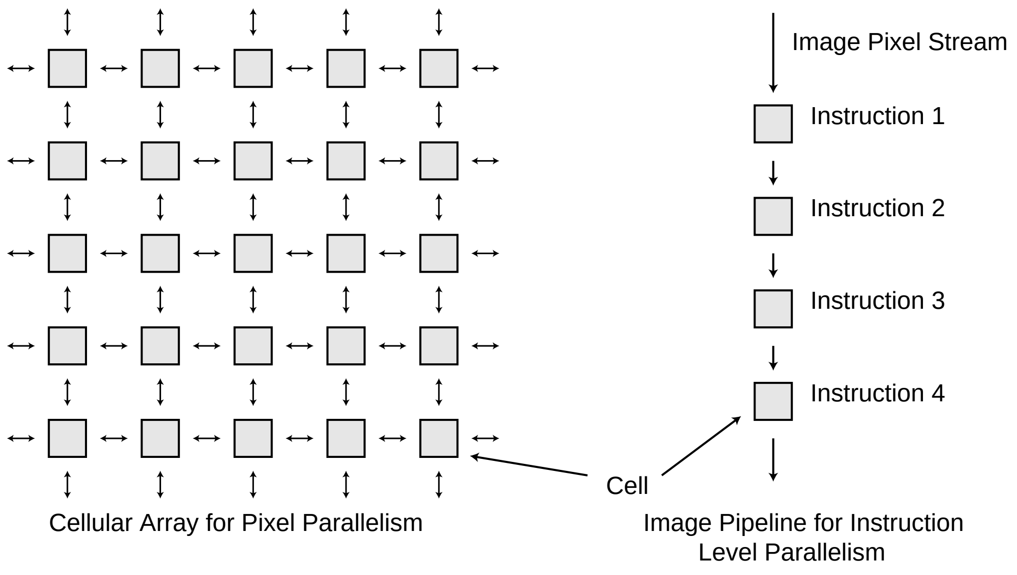


Image Pixel Stream
Instruction 1
Instruction 2
Instruction 3
Instruction 4
Cell
Cellular Array for Pixel Parallelism
Image Pipeline for Instruction Level Parallelism

Figure 35: Cellular Array (left) and Image Pipeline (right)

3.2.1 Cellular Arrays for Pixel Parallelism

Cellular arrays are a natural model for image processing [117] which exploit pixel parallelism. They consist of an array of cells in two, three or more dimensions. Each cell is associated with an image pixel and has local communication to neighboring pixels at all times. This high-bandwidth local communication is ideal for implementing spatial filters. Cellular Arrays are attractive due to their fine grain parallelism and local routing resources. This inherently parallel model is poorly suited to implementation on general-purpose microprocessor based systems. FPGAs can implement a programmable, maximally parallel implementation of a cellular array, but can only efficiently implement a small numbers of cells. Large arrays require multiple FPGAs and/or time multiplexing, and array initialization and result reading can soon dominate the computation time.

Within the context of a chromosome in an evolutionary algorithm, large-scale cellular arrays are generally not required. Learning is usually carried out with small training images and therefore only a small array is required. Cellular arrays are often applied iteratively. Iterations are equivalent to instructions or image-processing operators. Multiple instructions will take multiple clock cycles, but the instruction is applied to the entire image in 1 clock cycle. Chapter 4 describes maximally parallel implementations of several cellular architectures for AFE related problems.

3.2.2 Image Pipelines for Instruction-Level Parallelism

Another modular architecture used extensively in this thesis is the image pipeline. In this case only one cell, of the equivalent cellular architecture, is implemented. Input to the cell is through a continuous stream of pixels. They are supplied to the cell one pixel at a time, and usually in the case of images, in raster scan order. Although this is only one type of data arrangement, it is simple and naturally suited to real-time systems where input to the processor is often directly from a sensor. Speed-up is achieved by performing multiple instructions on the stream in parallel. This is illustrated on the right of Figure 35. After an associated latency, which is determined by the number of instructions, a pipeline is capable of producing a result every clock cycle. This means high-though put, and substantial speed-up compared to microprocessors can be achieved for multiple instruction pipelines.

Unlike cellular architectures, accessing a local neighborhood within an image pipeline must be carefully considered. All instructions in a pipeline are being executed at the same time, which can be demanding on memory bandwidth. Several ways of implementing spatial filters are described in Chapter 6.

The image pipeline is an effective way of dealing with large volume data. Figure 36 illustrates how this architecture is extended to include multi-spectral images by allowing multiple image channels as input. Cells read from one or more channels, perform some computation, and then output the result. Links between cells can also be considered images and are similar to *scratch planes* described within the GENIE system. This consistency in data flow is important when cells can be applied to either the original data cube or to outputs from other cells.

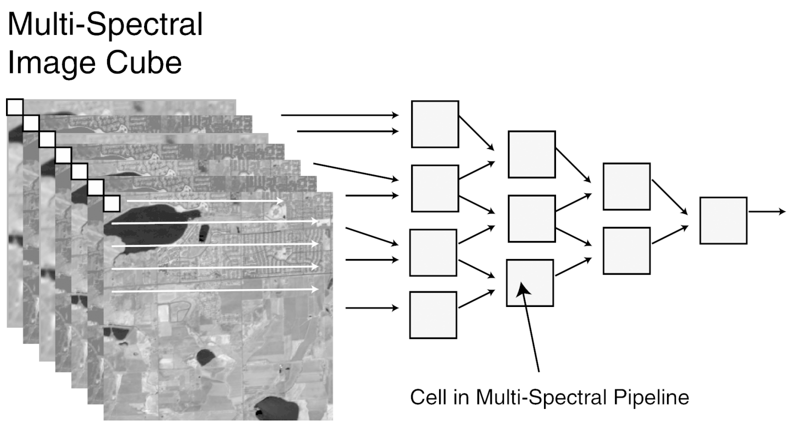


Figure 36: Multi-Spectral Image Pipeline

3.3 Chromosome Configuration

The Fitness Evaluator architecture has been described and abstract architectures for image processing chromosomes presented. The only question remaining is how a particular chromosome can be downloaded to the CC for evaluation. A first approach is suggested by the nature of FPGA device. SRAM FPGAs can be reprogrammed many, many times and therefore different bit-streams, which simply encode different chromosomes, could be downloaded each time. Unfortunately, most modern FPGA devices have large configuration bit-streams and therefore reprogramming the FPGA can take many milliseconds. Reconfiguration times in fact have increased and are likely to continue to do so as FPGAs reach higher and higher densities. Such times (1ms for the Virtex1000 FPGA) can often be large compared to the time required for fitness evaluation and therefore the bit stream is too slow to configure each individual in the population at run time.

Note, this is not the case for the Xilinx XC6216 rapidly reconfigurable FPGA and hence its popularity within EHW research groups. The techniques to be described are most useful to non-Rapidly Reconfigurable FPGAs, however the ideas are also applicable to the XC6200 series FPGAs.

Chromosome configuration can be considered a 2-stage process. This approach is motivated by the observation that a general purpose EA-accelerator needs to be configurable for:

(1) Different problems and
(2) Different chromosomes in the population for each problem.

For a particular problem, there is generally a specific representation and therefore search space. Each chromosome, in a particular EA experiment, is an instantiation of this representation. It is likely that two chromosomes from one experiment will have more in common than two chromosomes chosen from two different problems. In fact, for a particular EA experiments, chromosomes can have very similar implementation requirements. A key to efficient implementation is to localize where chromosomes differ so that reconfiguration time can be minimized. We present Figure 37 and the following terminology for clarity.

The *Generalized Chromosome* is implemented once, at the beginning of the EA run, using the FPGA programming bit-stream, and is common to all chromosomes for a particular EA problem. This is referred to as the 1st level of configurability. The *Generalized Chromosome* requires structure in order to localize differences and minimize reconfiguration times, and also flexibility to implement all EA chromosomes of interest. The second level of configurability is then used to rapidly fine-tune the *Generalized Chromosome*, to implement particular EA individuals. The result of the two-stage configuration process is a hardware implementation of EA chromosomes where fitness evaluation can be carried out at high speed.

Typically a Generalized Chromosome might define a set of operators, multiplexers and communication paths, which require perhaps 1 million FPGA configuration bits. The particular chromosome might then define things like the paths through multiplexers, and the specific value of filter coefficients which requires only a few hundred bits to define.

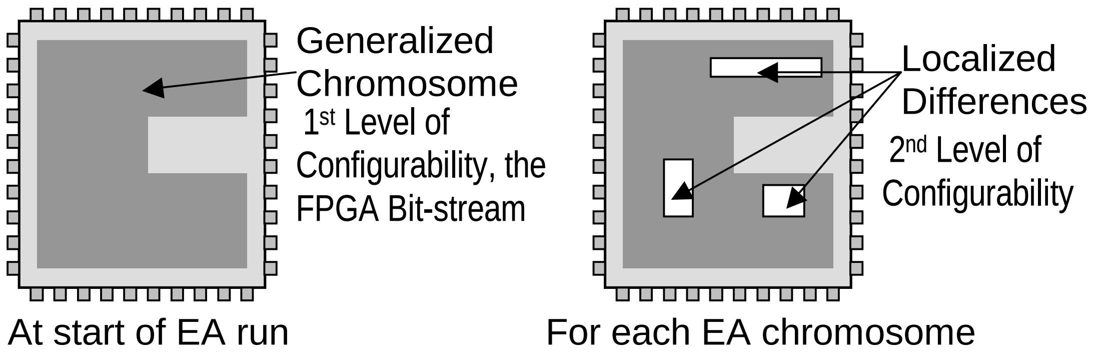


Figure 37: The Generalized Chromosome

The choice of *Generalized Chromosome* is important since it defines the EA search space, and therefore affects the EAs ability to find solutions to a given problem. A large proportion of this thesis is directed at finding suitable *Generalized Chromosomes* for AFE problems. How the second level of configurability is implemented depends on the device being used and is discussed further in the next two sections.

3.3.1 Rapid Reconfigurability

When using rapidly reconfigurable FPGAs such as the Xilinx XC6216, the 2nd level of configurability is also implemented with the FPGA bit stream. Partial Reconfiguration is used to fine-tune the *Generalized Chromosome* to implement particular chromosomes. The two-stage process is then:

1. A *Generalized Chromosome* is implemented at the start of the EA run with the FPGA bit-stream.
2. Particular EA chromosomes are then instantiated by using partial reconfiguration of the FPGA bit-stream.

Since fine-tuning of the *Generalized Chromosome* also involves the FPGA bit-stream, the functionality and connectivity of each FPGA logic block can vary from one chromosome to the next. This is considered hardware reuse at the gate level, which is explained further in the next section.

The two-stage configuration can be found in most FPGA implementations of EAs in literature, although it is often not identified. Work described in [107] used XC6264 FPGAs and implemented *Tree Templates* in the 1st level of configurability. *Tree Templates* were then instantiated with particular chromosomes with Partial Reconfiguration.

3.3.2 Alternatives

Modern high density FPGAs have moved away from memory mapped rapidly reconfigurable architectures seen in the XC6200 series. A second level of configurability must therefore be implemented within the *Generalized Chromosome* itself. Essentially, this involves increasing the complexity of design to include on-chip configuration registers. These configuration registers then set particular control lines in the design that dictate arrangement and precise behavior of problem specific hardware components. An example of this approach can be seen in Figure 38.

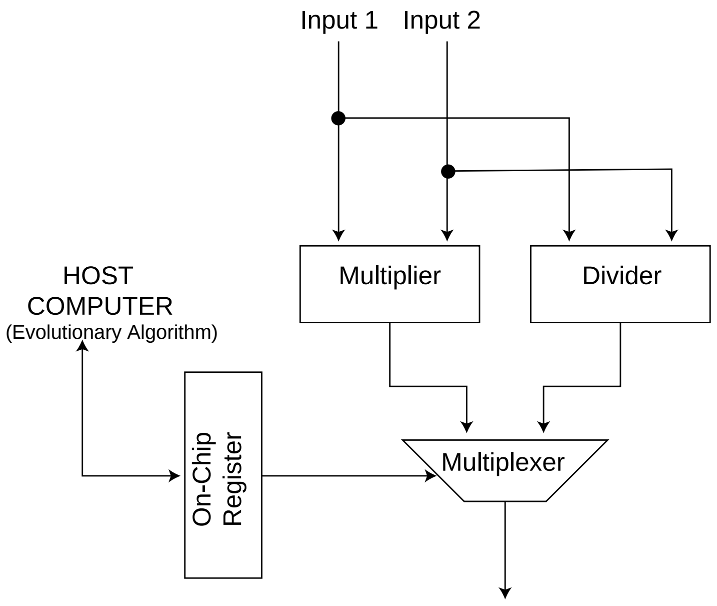


Figure 38: Example of the 2nd Level of Configurability

In this example, the designer has decided that the EA should optimize the type of operation, either multiplication or division, that is applied to two inputs. Both multiplier and divider are therefore implemented and supplied with the necessary inputs. A single bit can represent the choice of operation. This bit is stored in an on-chip register that can be accessed by the host processor at high speed. The on-chip register controls a multiplexer, which selects either the output from the multiplication or the output from the division.

In this example it can be seen that control lines and multiplexers can soon dominate hardware resources. The main design consideration with this technique is therefore hardware reuse from one chromosome to the next. Hardware reuse between chromosomes maximizes the amount of FPGA resources being used at any one time leading to increased performance. Several ways of implementing hardware reuse are now described.

Gate Level Reuse

Rapid Reconfigurability means the FPGA has hardware reuse at the gate level. Such fine-grain flexibility can also be incorporated in more conventional devices. Virtex FPGAs from Xilinx, allow 2 look-up-tables to be configured as a dual port RAM. One port can be tied to the host and used to program the RAM. The second port can then be used in the design to implement user logic. Note, to implement a reconfigurable look-up-table, the resource cost is effectively doubled (2 LUTs instead of 1). However, if the number of reconfigurable LUTs required is small, this technique can be useful. This technique was used to implement reconfigurable stack filters described in *Everything on the Chip* [115].

Arithmetic Level Reuse

Many EA experiments do not require the fine-grain flexibility of Gate-Level reuse. In this case, hardware reuse can be implemented in the *Generalized Chromosome* at a much higher level. Generalized pipelined arrays, suggested by Kamal in [118] are an excellent example of how such flexibility can be implemented. In this work, a pipeline is proposed that can implement a number of common arithmetic operations such as multiplication, division and square root by setting control lines. These operations have very similar hardware requirements and therefore can be combined with large area cost savings. An arithmetic array was synthesized for an Altera Flex10K FPGA (not described) and details are summarized in Table 1 where numbers represent Altera Logic Blocks. Although this device is not used for the remainder of the thesis, the relative cost of the different arithmetic operations would be similar for

Xilinx Virtex devices. This is due to the similar level of functionality that is provided by the Altera and Xilinx Virtex logic blocks (coarse grain devices).

Operator	*****	**/**	**SQRT**	**Generalized Arithmetic Array**
Estimated Area (Logic Blocks: Altera Flex 10K)	206	286	305	429
Resource Gain				1.98

Table 3: Resource Estimates for a Generalized Arithmetic Array

Through reuse of hardware, the computational resources available to the EA are effectively doubled. This is considered hardware reuse at the arithmetic level. These arithmetic arrays can be useful building blocks in many EA applications.

Operator Level Reuse

Hardware reuse can be incorporated at any part of the implementation. A particularly useful approach to *Generalized Chromosome* design in this thesis was hardware reuse at the operator level. Image processing has many computationally similar operators, such as smoothing, edge detection, convolution and morphology. Chapter 6 describes how flexibility can be incorporated into the *Generalized Chromosome* to implement a number of these operators using the same hardware resources.

3.4 Chapter Summary

The major design considerations for implementing EA experiments on CC have now been described. All experiments that follow use the single chromosome Fitness Evaluator architecture.

The chromosomes evolved in Chapter 4 are examples of cellular arrays. The Rapidly Reconfigurable XC6216 FPGA is used and partial reconfiguration exploited to rapidly fine tune the Generalized Chromosome.

In subsequent chapters, chromosomes are image pipelines. This enables investigation of more practical AFE problems and application to large volume multi-spectral image cubes. In these chapters, the Virtex FPGA is used and partial reconfiguration is implemented by increasing the complexity of the Generalized Chromosome.

This chapter has presented hardware design practices used by many hardware engineers. The novel contribution of this chapter is the application of these design practices to EA implementations. In particular:

- A clear picture of the EA co-design spectrum in terms of host / CC communications was presented.
- A two-stage configuration technique, seen in almost all EA implementations reported in literature, was formalized in terms of a *Generalized Chromosome*.
- Hardware reuse was identified as the main design consideration for EA experiments on non-Rapidly Reconfigurable FPGA devices. Several mechanisms for hardware reuse were suggested.

Chapter 4

Evolving Cellular Arrays

In this chapter, the chromosomes considered are maximally parallel cellular arrays. This means that a cell, or processor is implemented for each pixel in a small training image. Due to the finite resources available in FPGAs (particularly the XC6216 used in this chapter), each cell in the array is limited in what it can implement. This makes solution of practical AFE problems with maximally parallel implementations difficult. The appeal of cellular arrays is the promise of massively parallel systems, with little or no centralized control. The fact that resources are limited also promotes the development of simple solutions, which is an excellent way to approach hardware design in general. Finally, it is believed the cellular approach will become increasingly important as circuit densities and FPGA capacities increase [119].

Section 4.1 describes the host-CC communication requirements for maximally parallel fitness evaluation. Section 4.2 then describes implementation of perhaps the simplest cellular array, cellular automata (CA). The experiment in this section is implemented on the *Hot Works* CC based on the rapidly reconfigurable XC6216 FPGA. A *hybrid* XC6216/Cellular Automata[4] model is evolved to solve a binary pattern classification problem. This experiment demonstrates how the combination of hardware resources and software model can lead to unique search spaces which can be efficiently implemented and better suited for particular problems than the raw FPGA bit-stream.

In Section 4.3, a novel approach to gray-value texture classification using Stack Filters is described. The approach is made possible by the flexibility of EA

[4] The Cellular Automata model is dictated by the resources available on the XC6216 FPGA

optimization techniques. Stack filters can be implemented very efficiently in CC and therefore may be suitable for maximally parallel implementations.

4.1 *Maximally Parallel Fitness Evaluation*

FPGA resources are usually homogeneous, and spatially distributed and therefore naturally suited to maximally parallel implementations. In this case, significant speed-up can be achieved for chromosome execution. However, since pixel processors are spatially distributed across the FPGA, both loading training data and calculating the fitness metric is more complex.

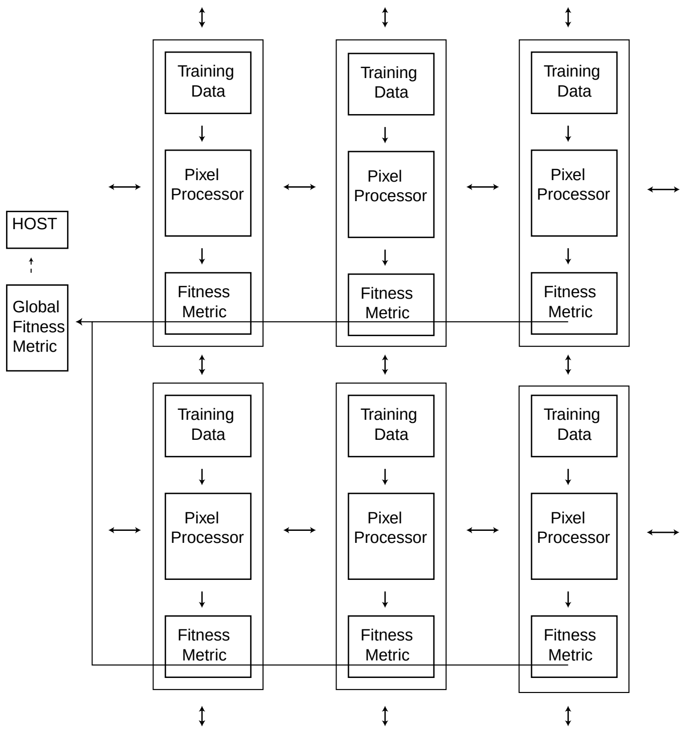


Figure 39: Ideal Host/RCC Architecture

Figure 39 illustrates the ideal architecture for maximally parallel implementations. In this case, the training data and the fitness metric are implemented local to the pixel processor. Since the array exploits pixel parallelism, it is desirable that training data be supplied to all pixel processors at the same time. This would require large bandwidth to a centralized memory and therefore is best implemented in small, distributed memories across the array. The fitness of an image-processing algorithm is often based on a sum of distance measures for each pixel. The distance measure for each pixel can be implemented at the output of the pixel processor. Global communication is then required to combine these measures to produce a single fitness score. If this can be implemented in CC, the CC-to-Host communication is minimized.

Due to resource limitations of the XC6216 FPGA, the experiments in this chapter use a different architecture, which is illustrated in Figure 40. Dashed lines in Figure 40 indicate the host communication with the array. In this case, the communication requirements are greater. Training data is stored on the host computer, and must be sent to the CC for each chromosome evaluation. The Fitness Metric is implemented on the host and therefore the final output of the array must be sent back to the host for each chromosome. The advantage of this approach is that resources do not have to be used to implement an on-chip Fitness Metric. This means more pixel processors can be implemented and therefore greater pixel parallelism can be achieved.

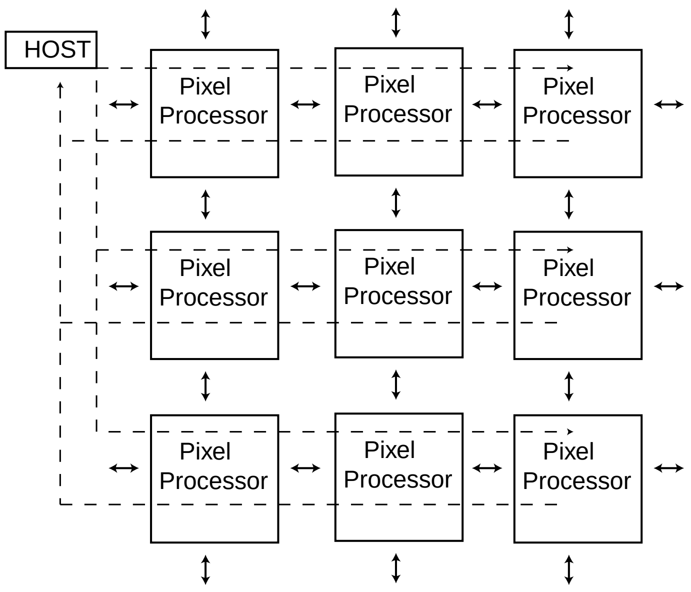


Figure 40: Actual Host/RCC Architecture

4.2 Evolving Cellular Automata

Cellular Automata have developed under a number of different names from a variety of fields. John von Neumann is known to have developed the idea of cellular automata in the late forties to model complex, extended systems [120]. In their simplest form, Cellular Automata can be considered a homogeneous array of cells in one, two, three or more dimensions. Each cell has a finite discrete state. Cells communicate with a number of local neighbors and update synchronously according to deterministic rules. The rules usually employ:

- Spatial locality: A cell updates its state depending on the state of its surrounding cells (referred to as a neighborhood)
- Temporal locality: A cell updates its state depending on the state of itself and neighbors a small number of time steps in the past (usually one).

Figure 41 illustrates the process for a one-dimensional CA, where each cell can have one of two possible states (a binary CA) and each cell communicates with its immediate neighbors. Figure 41 also includes an example rule table for this CA. The bold entry corresponds to the update rule used to take the highlighted cell or state 0 to state 1 in the next time step. The number of neighbors that a cell communicates with is called the CA radius. In the case of Figure 41, the CA has a radius of 1. The number of states that a cell may reach is not limited but must be finite. The state of every cell within a CA at any particular moment in time is often referred to as the CA configuration [121].

State	*Left*	*Right*	*New State*
0	0	0	0
0	**0**	**1**	**1**
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

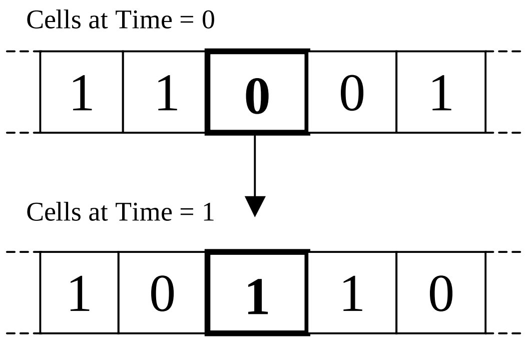


Figure 41: One Dimensional Cellular Automata and Rule Table Example

CA's can produce a wide variety of complex behavior. The CA structure is simple and easily definable and much time has been spent on classifying CA global (or emergent) behavior. Wolfram in [121] has exhaustively investigated one-dimensional binary CA's with a radius of 1. The exponential increase in rule table combinations for CA's of higher dimension and larger states however has meant that CA behavior is still largely unexplored.

Several researches have used Evolutionary Algorithms to find cellular automata rule tables that produce *useful* behavior. Melanie Mitchell and James P. Crutchfield of the Santa Fe Institute explored application of GA to one-dimensional binary CA to perform a density classification task [122].

Most relevant to experiments of this chapter are works by Sahota, which evolve cellular automata to perform binary image processing. Figure 42 is an example of the types of training images that Sahota used in [123]. The CA is first initialized with an input image, examples of which can be seen on the left of Figure 42. The fitness of a particular CA rule table is based on how closely the CA array matches the desired output image (shown on the right of Figure 42) after a predefined number of iterations.

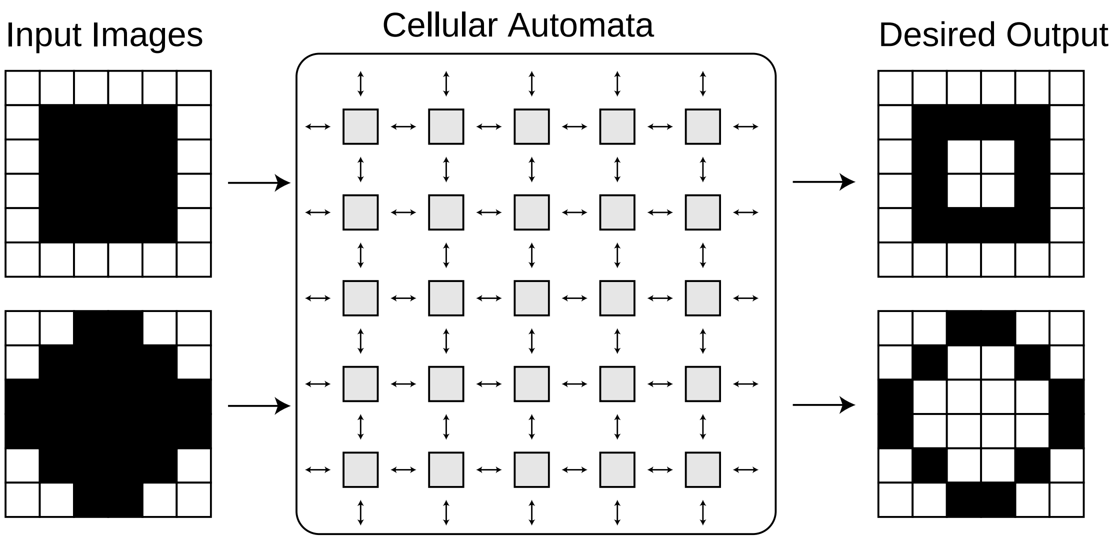


Figure 42: Example of Training Images used in Experiments by Sahota [123].

The desired behavior can be seen to be binary edge detection. In [124], Sahota extended the system to include multiple CA layers between input and output. He found that this was of benefit to more difficult problems where noise was introduced.

In the next few sections, an FPGA based experiment is implemented that is similar to Sahota's work. This experiment is a demonstration of principle. It is also motivated by the potential of significant speed-up compared to software for CA experiments. Additionally, two CA chromosome representations are investigated: One is inspired by evolvable hardware where EA have been applied directly to the FPGA bit-stream.

The second introduces greater constraints on the search space and is representative of the approach taken in subsequent chapters.

4.2.1 The hybrid XC6216 Cellular Automata

FPGAs can be used to efficiently implement maximally parallel CA if the model is small enough. One of the simplest CA models has 5 neighbors (the cell itself, and neighbors above, below, left and right) and 2 states. To implement this CA, each cell must be able to implement any logic function of 5 variables. This can be achieved with a look up table, or alternatively with a logic tree illustrated in Figure 43.

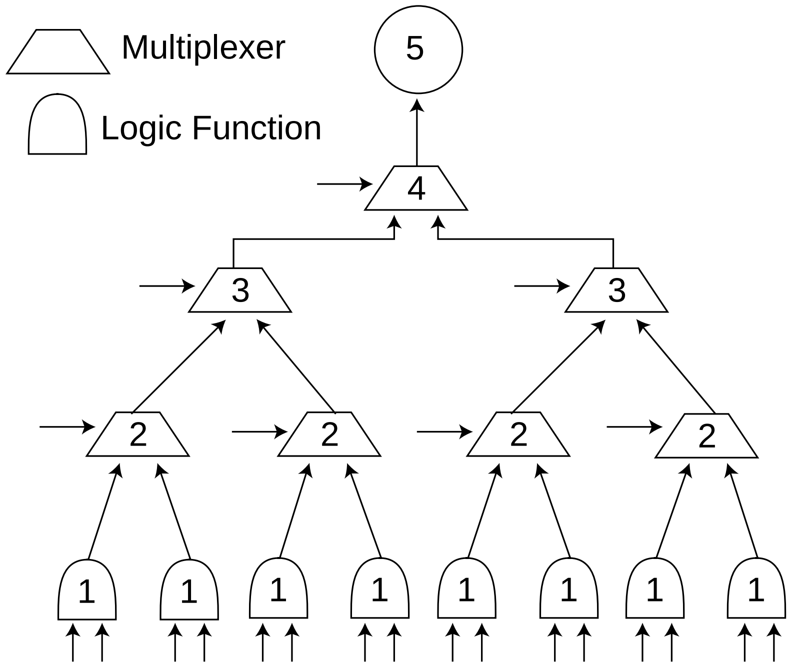


Figure 43: CA Logic Tree

This logic tree can be implemented in a 4 by 4 group of XC6216 logic blocks. This is illustrated in Figure 44. Solid arrows illustrate the logic block outputs, which implement the logic-tree structure. The CA cell state is stored in logic block 5. This logic block output is routed back throughout the 4 by 4 group and to all north, south, east and west faces of the cell. Dashed arrows illustrate its path through the local routing resources.

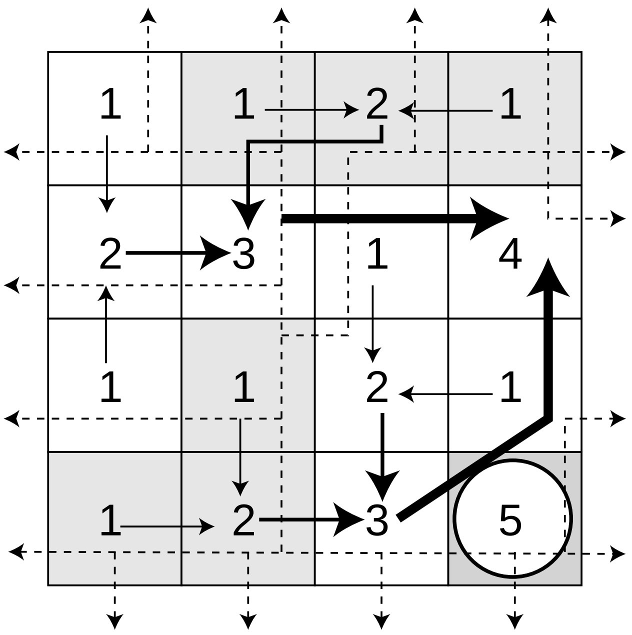


Figure 44: Implementation of Logic Tree in 4by4 group of XC6216 logic blocks

State information from a CA cell's north, south, east and west neighbors are routed across all logic blocks through the XC6216 length 4 interconnects. This means that every logic block has access to all 5 Boolean state variables. The 4 by 4 group of Figure 44 is replicated across the XC6216 to form a 15 by 15 two-dimensional Cellular Automata. This is illustrated in Figure 45. The North/South and East/West ends of the array are wrapped around via the XC6216 chip length routing resources.

Codd in [125] proved that a 2-state 5 neighbor CA capable of universal computation did not exist with a finite initial configuration. Several authors achieved computation-universal CA models by adding more states or larger neighborhoods [126]. In this experiment, the 2-state 5-neighbor CA is extended by including the memory resources (flip-flops) that are available in each XC6216 logic block. This leads to a CA model based on neighborhood information from further back in time. With this extension, it is possible to demonstrate the computation universality of this *hybrid* XC6216 CA model by simulating Minsky's two-register machine [126].

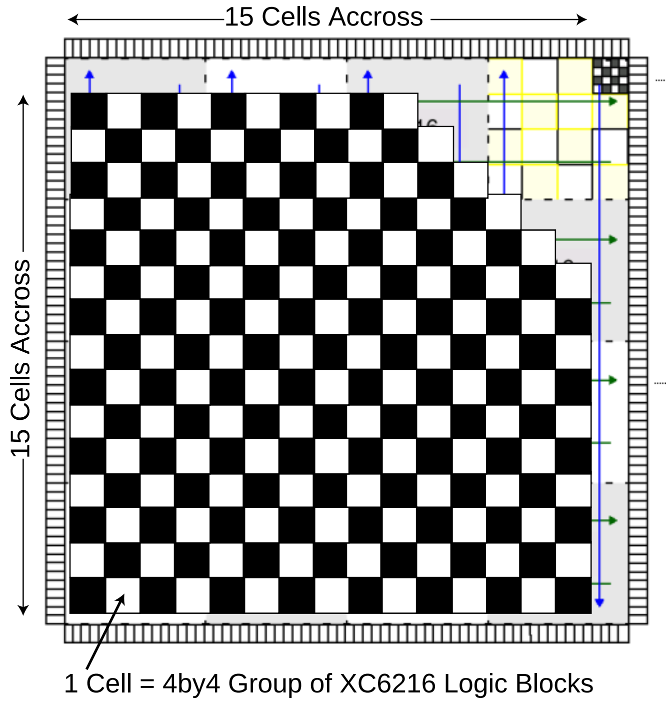


Figure 45: Tiling CA cells across the XC6216 FPGA

4.1.1 Host Program

At the start of the evolutionary experiment, the host program configures a *Generalized Chromosome* that remains constant throughout the EA run. In this case the *Generalized Chromosome* corresponds to predefined routing between 4by4 groups that implements the 15 by 15 array of Figure 45. During evolution, partial reconfiguration is used to configure the precise functionality of logic gates and multiplexers of Figure 43. All cells in the array can be configured using Partial Reconfiguration at the same time. This is possible due to a feature known as *Wildcard Registers* [85] that are implemented in the XC6216. These registers allow the configuration address space of the XC6216 to be written in multiple locations with one memory access. Note, this feature can only be effectively used if the design is homogenous across the XC6216 device. This is the case with maximally parallel implementations of CA.

Due to the limited size of the array, each chromosome in the experiment to come is applied to a number of training images, so that its fitness can be determined accurately. The main genetic algorithm loop and genetic operators are implemented on the host processor. The host/CC communication for evaluating a chromosome is illustrated in Figure 46.

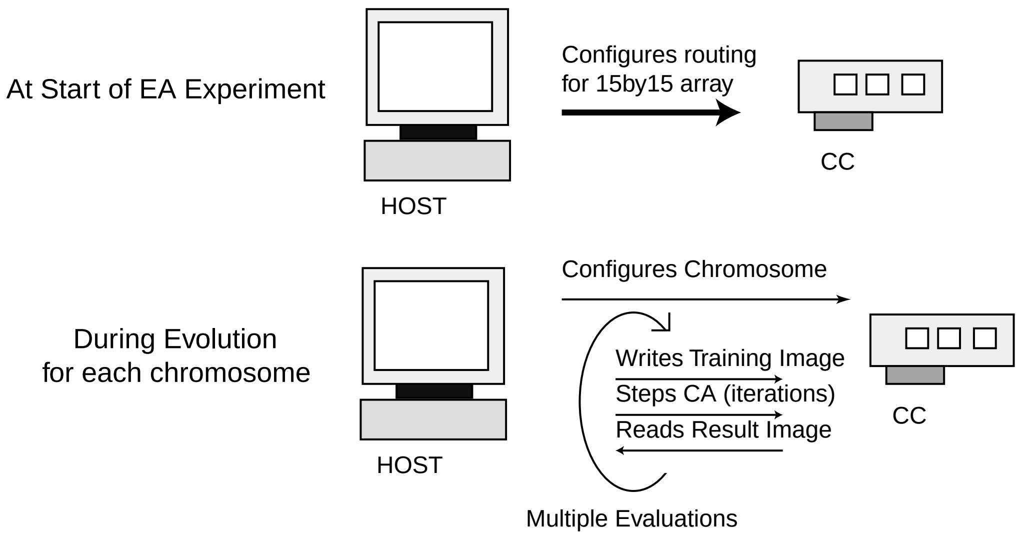
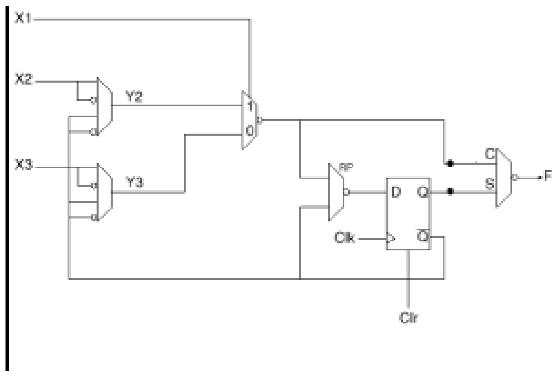
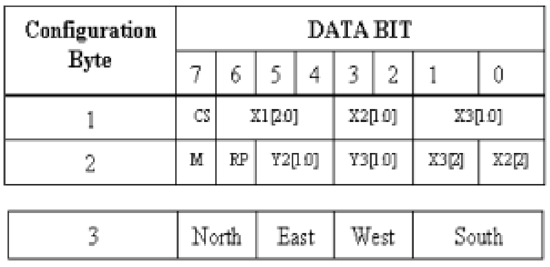


Figure 46: Host/CC Communication Required for Chromosome Evaluation

4.1.2 Representation and Schedule

When cellular automata are evolved in software, the chromosomes are usually fixed size rule tables. An example of a chromosome would be the right column of Figure 41. The *hybrid* XC6216 CA suggests alternative representations that are easier to implement, and may also lead to smaller, more relevant search spaces for particular problems.





Configuration Byte	DATA BIT							
	7	6	5	4	3	2	1	0
1	CS	X1[2:0]			X2[1:0]		X3[1:0]	
2	M	RP	Y2[1:0]		Y3[1:0]		X3[2]	X2[2]
3	North		East		West		South	

Figure 47: a) XC6216 Function Unit **b) Configuration bytes**

The function unit in each XC6216 logic block is illustrated in Figure 47a, and is defined by three eight bit configuration bytes shown in Figure 47b. In the first two configuration bytes, CS defines whether or not the function unit will make use of the flip-flop resource. The X1, X2 and X3 configuration bits define the input signals to the logic gate or multiplexer. Y2 and Y3 define the functionality of the gate or multiplexer. RP sets the flip-flop as read-only and M defines additional routing resources.

Configuration byte 3, illustrated in Figure 47b, controls the local multiplexers to neighboring blocks. This essentially dictates the connectivity between logic blocks, and therefore the potential inputs to the function unit. This can be more easily understood by referring to Section 2.3.1.1 in the Literature Review. A more detailed description of the configuration bytes can also be found in [85]. Two different hardware representations are investigated:

FPGA Bit-string: The first, inspired by EHW, is based on configuration bytes for the 4 by 4 group of function blocks that make up a cell. The connectivity between each 4 by 4 group is pre-defined by the *Generalized Chromosome* that implements the 15 by 15 array. All routing and functionality within the 4by4 group is included in the representation which is completely specified by the 3 configuration bytes / logic block. The chromosome length for the *FPGA Bit-string* representation is therefore 48 bytes (4 by 4 group by 3 bytes) or 384 bits.

Logic Tree: The second representation is based on the logic tree of Figure 43. In this case the routing resources within each 4 by 4 group of function blocks is pre-defined by fixing the 3rd configuration byte of Figure 47b. This means the 3rd configuration byte is included in the *Generalized Chromosome.* It maintains the logic tree structure between chromosomes and only the functions and multiplexers at each node are allowed to evolve. A CA chromosome is therefore specified by 2 configuration bytes / logic block. The effective chromosome for the *Logic Tree* GA is therefore 32 bytes or 256 bits.

4.1.3 Experiment

A performance comparison of the *FPGA Bit-string* and *Logic Tree* hardware representations, as well as a software based CA rule table representation is made. A Genetic Algorithm with Elitism described in Section 2.2 of the Literature Review is used. The reproductive schedule is summarized in Table 4.

EA Parameter	**Number**
Population Size	300
Parents	100
Number of Generations	100
Reproduction	
Elite	100
Crossover and Mutation (parents: 80 %)	200

Table 4: EA Schedule for Cellular Automata Experiments

Crossover

- For the *FPGA Bit-string* representation, a one-point crossover is applied to the 384-bit configuration bit string.
- For the *Logic Tree* representation a genetic programming type crossover that constrains the structure of CA offspring to the logic tree of Figure 43a is used. A crossover node is randomly selected and the corresponding sub-trees are swapped between parent chromosomes.

Mutation

- In both cases, the crossover offspring are mutated in up to 4 randomly chosen positions.

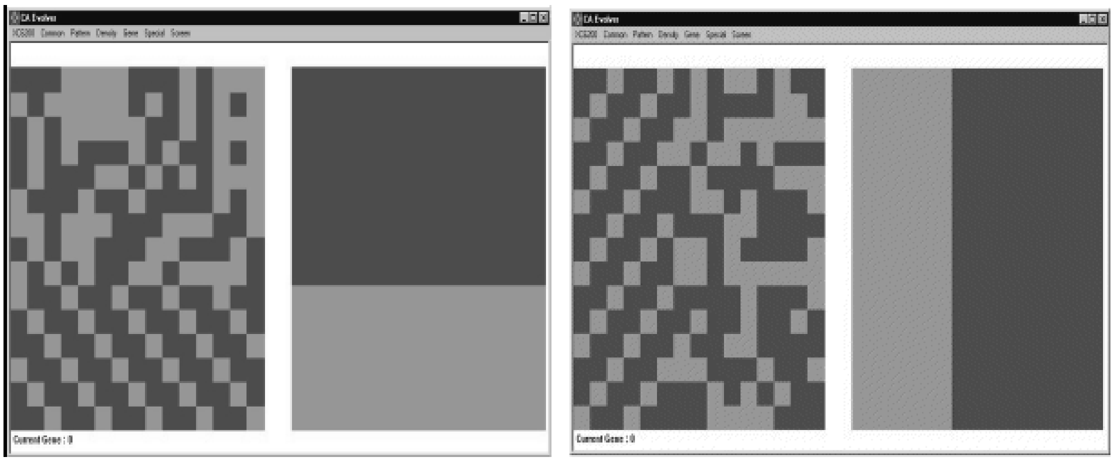

CA Evolver
XC6200 Common Pattern Density Gene Special Screen
Current Gene : 0
CA Evolver
XC6200 Common Pattern Density Gene Special Screen
Current Gene : 0

Figure 48: a) Horizontal and b) Vertically Segmented Training Images

The XC6216 CA is applied to a binary texture classification problem, which involves identifying a particular pattern within the 15 by 15 pixel area. The binary pattern used in the experiment is diagonal lines. Two screenshots of the CA Evolver host program, each with training image on the left and desired output on the right, are depicted in Figure 48. Each training image is divided either horizontally as in Figure 48a, or vertically shown in Figure 48b. The non-patterned segment is filled with random pixels, of a density randomly chosen from a uniform distribution between 0 and 1.

The state of each CA cell is initialized with the corresponding point in the 15-by-15 training image. The XC6216 is then clocked for 200 cycles at 33Mhz. This corresponds to 200 iterations of the CA array. The fitness of the CA individual is calculated by a bit by bit comparison of the state of CA cells, after 200 iterations and the desired output. A fitness counter is incremented for each CA cell in correspondence with the ideal image. While the counter is greater than zero it is also decremented for each cell that is not in correspondence. As in [123], CA individuals that lead to a collapsed image (the state of the CA array ends up all 1's or all 0's) are penalized and receive a fitness score of 0.

Each chromosome within the population is applied to 300 training images for each evaluation. The overall fitness for a chromosome is calculated as the root mean square over the 300 images. The elite chromosomes are re-evaluated each generation to ensure they do not exploit bias that may exist within a given 300 images. Due to the large number of training images that each chromosome is exposed to, testing chromosomes on additional images to estimate generalization performance was not considered necessary.

4.1.4 Results and Discussion

A software-based experiment was also implemented to compare execution times and estimate speed-up. The chromosome in this case was the more typical 5-input, 2 state, look-up-table. The computation time for the Evolutionary Algorithm can be decomposed according to Equation 10.

$$T_{total} = T_{init} + \text{Generations} * (T_{fit} + T_{ga}) \quad \textbf{(10)}$$

T_{init} is the time to generate a population and training images. T_{fit} is the time to evaluate the fitness of the population and T_{ga} is the time to apply genetic operators between generations.

Measurement in Seconds	Software Rule Table	FPGA Bit-string	Logic Tree	Relative Speedup
T_{init}	0.1	0.1	0.1	1
T_{fit} per Generation	1987.9	34.22	34.17	58
T_{ga} per Generation	0.05	0.06	0.08	1

Table 5: Estimation of Speedup

Table 5 summarizes the execution times for the various components. A speed-up of 58 was obtained compared to the software implementation on a Pentium II microprocessor running at 233MHz. More significant performance gains would be possible if a larger FPGA device was used. Xilinx offers a XC6264 FPGA that has approximately 4 times as many resources as the XC6216 used in this experiment. This would enable a 30 by 30 CA maximally parallel CA to be implemented. Execution times could be expected to be slightly more than those reported in Table 5, since array initialization and result reading would be more expensive.

A qualitative comparison of the two FPGA based representations, *FPGA Bit-string* and *Logic Tree*, as well as the software based rule table representation was made. For each experiment, the GA had 100 generations in which to find a CA to correctly classify the diagonal line pattern. Figure 49 illustrates the average pixel error (number of pixels misclassified) as a function of GA time (generations).

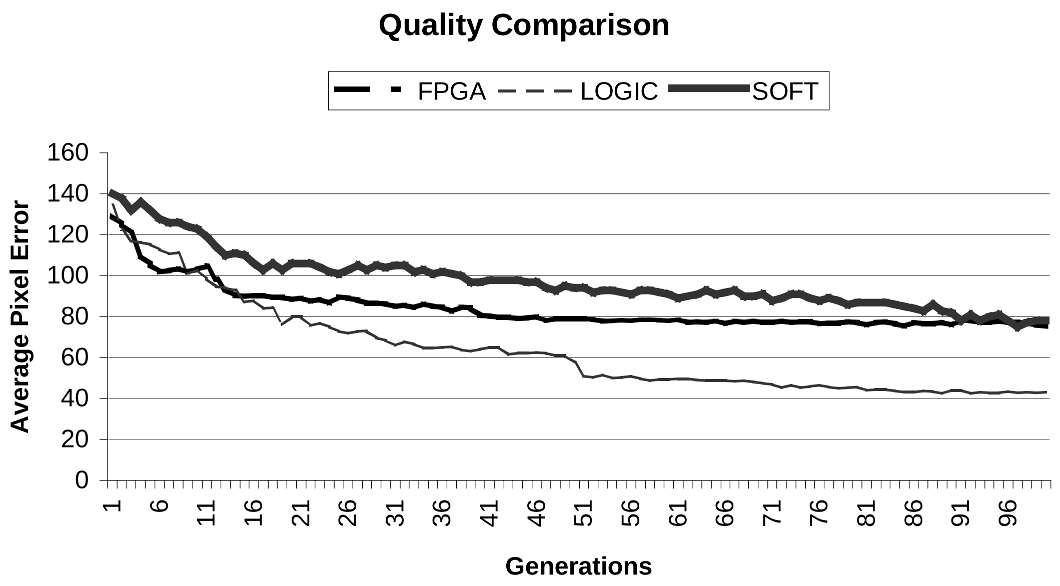


Figure 49: Comparison of GA performance for 3 representations

It is observed that both FPGA representations appear to have converged faster than the software rule table representation, and it is possible that further evolution would lead to lower pixel error solutions for the software rule table representation. However, a more important observation to this thesis is the fact that the *Logic Tree* representation outperformed the *FPGA Bit-string* representation in all experiments. It is hypothesized that constraining connectivity and evolving only function avoids strange feedback loop/analog circuit behaviors and allows a more detailed exploration of a smaller, more relevant search space.

4.1.5 Problem Specific Constraints

In several runs, individuals of note appeared that performed well for images segmented in one direction (eg. vertically) but poorly in the other direction. To encourage the EA to find solutions that performed well in both directions, the *Logic Tree* search space was further constrained. The multiplexer selector of node 4 in Figure 43 was pre-defined to take input from the cell's current state. This reduces the size of the non-symmetric rule space available to the EA and is illustrated in Figure 50.

Also, due to the close spatial relationship between training and desired output images, two nodes of type 1 (one in each half of the logic tree) were also pre-defined to store the initial training image. This means the initial configuration of the array is available even after many CA iterations. This is also illustrated in Figure 50. The best of run CA from these experiments had an average pixel error of 40 pixels, lower than any other CA found.

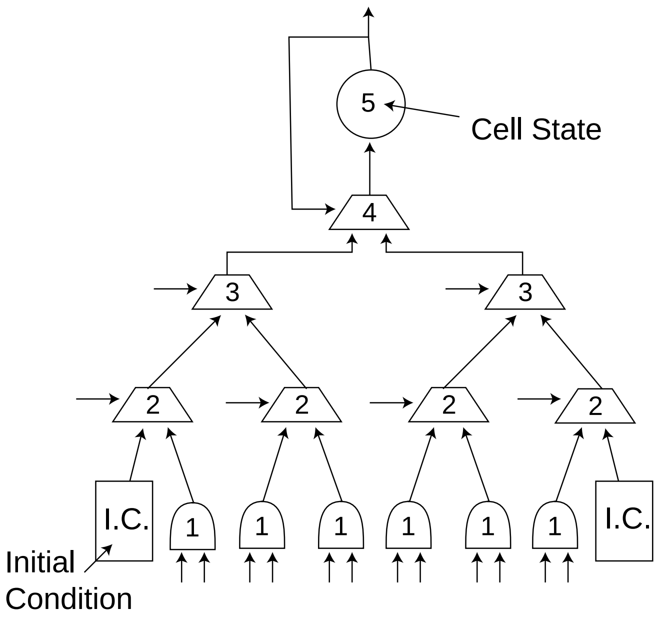


Figure 50: Introducing Problem Specific Constraints

4.1.6 Summary

By constraining connectivity within each CA cell, the *Logic Tree* representation usually produced better solutions than the *FPGA bit-string* representation. Problem specific constraints could also be implemented more easily with the *Logic Tree* representation than with the conventional software rule table representation (although such constraints are certainly possible). This can be attributed to the 'program like' nature of the *Logic Tree* representation. While the rule table representation abstracts the internal workings of the CA algorithm, the *Logic Tree* allows fine grain decomposition of CA. Therefore, there is greater visibility into the CA's algorithmic components, and constraints are more easily implemented.

4.2 Evolving Stack Filters

To use cellular automata (CA) for gray-valued image processing is a difficult problem since rule-table size grows exponentially with the number of states. In this section a novel alterative using Stack Filters [9] is investigated. Stack filters are closely related to cellular automata. In the binary case, stack filters represent a subset of cellular automata rule spaces known as Positive Boolean Functions (PBF). Properties for which they are named, threshold decomposition and stacking, allow stack filters to be applied to gray-valued images with little increase in hardware complexity.

Stack filters have been used widely in image and signal enhancement, however they have not been previously applied to feature classification. Experiments in this section investigate the feasibility of using stack filters for a texture classification problem. The potential of this approach lies in extremely efficient implementation of gray-valued cellular arrays. However, since CC design time is significantly greater than software, the experiments of this section were carried out in software.

4.2.1 Introduction to Stack Filters

Stack Filters are a class of non-linear filter that include the median filter as well as many of the fundamental morphological filters such as erosion, dilation. These filters are usually applied as a spatial filter and are known to be well suited to hardware implementation. This is due to threshold decomposition and stacking properties, which allow gray-valued images to be processed with bit-level hardware [127]. This property is illustrated in Figure 51 for the median filter.

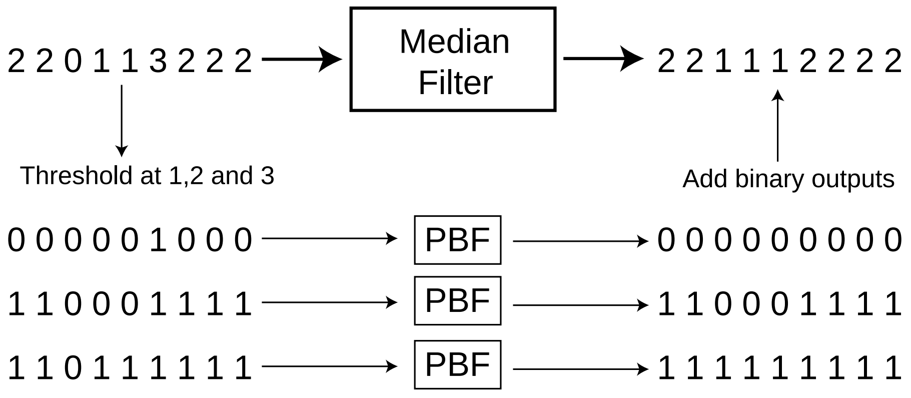


Figure 51: Threshold Decomposition of Median Filter

A stack filter is uniquely defined by a positive boolean function of its inputs. Positive Boolean Functions (PBFs) are a subset of Boolean logic functions in which no input may be negated. The PBF is applied to each level of the threshold decomposed input, and the output levels are then summed (made possible by the stacking property) to produce a gray-valued output.

While efficient hardware implementation is the primary motivation, there are several reasons why stack filter architectures may have properties desirable to texture classification problems. Stack filters are generalizations of the median filter, which has excellent noise removal properties. Also, an essential requirement of a good texture classification algorithm is insensitivity to image illumination. The stack filter operates on relative, order statistical information and is therefore is not affected by differences in image illumination. Stack filters are also closely tied to Mathematical Morphology [14]. By evolving the shape of a stack filters neighborhood, structural information can be extracted which is a useful way to characterize texture.

Stack Filter Implementation

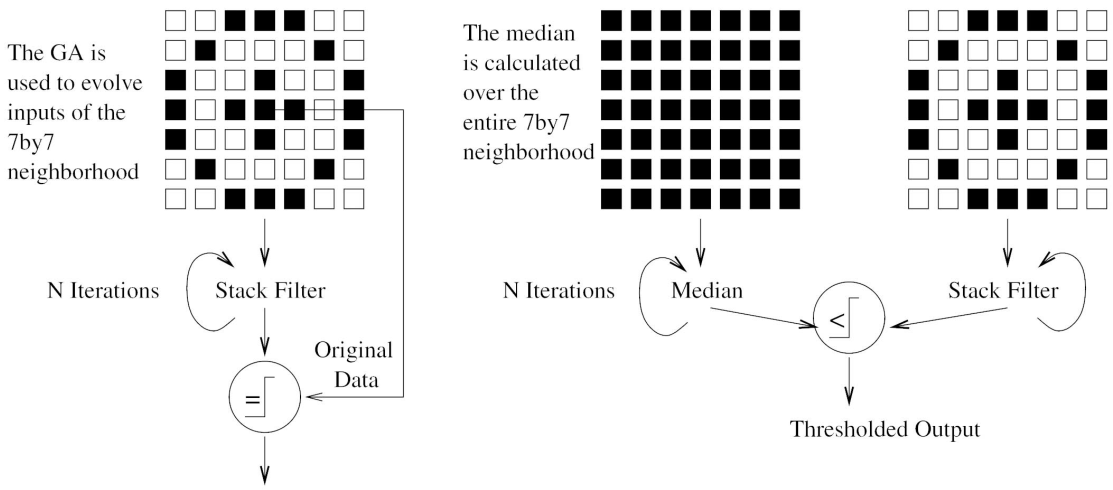
When implementing maximally parallel cellular arrays, only extremely simple pixel processors allow sufficient densities of cells for practical applications. Binary cellular automata are an excellent example of simple pixel processors. The stack filter architecture in Figure 51 at first appears to have complexity proportional to the number of levels in the input signal. This illustration of stack filters, although useful for understanding the threshold decomposition process, is not what is usually implemented and extremely efficient mechanisms have been proposed [128].

Most useful to maximally parallel implementations is a bit-serial implementation of stack filters [127]. This has very similar hardware complexity to binary cellular automata. Threshold decomposition is a function of time and gray-valued image processing can be implemented with binary components. The bit-serial implementation is described and used in *Everything on the Chip* [115] in Appendix B.

4.2.2 Thresholding Stack Filters and Fitness Evaluation

When applying stack filters to classification problems, it is important to realize that the stack filter output will always be one of the neighborhood samples. This means that if the feature of interest lies in a mid-valued range compared to the input signal dynamic range, the stack filter output over the feature of interest will not be linearly separable from non-feature. At the very least a banded threshold is required in order to separate the data. Two alternative ways of thresholding were explored.

The first approach is inspired by the associative memory nature of stack filters described in [129]. With this approach an 'invariance threshold' is applied before calculating a distance measure. The threshold is *high* if the stack filter is invariant over an input sample and low otherwise. By minimizing the distance between this thresholded output and the target classification, a stack filter is evolved whose invariant signal set exists more in the true samples than in false samples. This threshold can be implemented very easily in hardware and is illustrated on the left of Figure 52.



The GA is used to evolve inputs of the 7by7 neighborhood
N Iterations
Stack Filter
Original Data
=
The median is calculated over the entire 7by7 neighborhood
N Iterations
Median
<
Stack Filter
Thresholded Output

Figure 52: Thresholding strategies applied to stack filters.

The second approach is inspired by the 'generalized median' interpretation of stack filters. The stack filter output is thresholded by a neighborhood median. For a given input size there is partial ordering of stack filters that was described in detail in [130].

What has not been explored is the ordering of stack filter outputs from filters using different sized neighborhoods. In this experiment, the ordering required is defined (the output of the stack filter must be equal or higher to the median over the feature of interest, and less than the median for non-feature) and the EA is used to search for stack filters that satisfy this criteria. Figure 52b illustrates the 'median threshold' described. Note, that by thresholding with a median the hardware requirements are effectively doubled.

Iterative application of stack filters has been used to improve performance and several researchers have investigated stack filter convergence behavior. Some classes of stack filters have been shown to possess the convergence property and upper bounds for convergence times have been derived. Other classes of stack filters do not possess the convergence property and have been shown to produce oscillatory outputs [131]. As illustrated in Figure 52 both stack filter and median, are applied N times before thresholding occurs. Several different values of N were investigated.

Target classifications are provided with the training data in the form of binary masks, which define both feature (true) and non-feature (false) for each texture class. The fitness of a classification algorithm can be calculated with a distance measure (typically Euclidean or Manhattan) between the algorithm output and the binary mask.

The distance measure used is a weighted hamming distance. True and false classes contribute equally to the score, which is defined in Equation 11. T_c and F_c represent the number of true and false pixels correctly classified and T_T and T_F are the total number of true and false pixels respectively. A perfect classification will result in a score of 1000. Further discussion of this fitness metric can be found in Chapter 5.

$$Fitness = \left({}^{T_C}\!/_{T_T} \right) * 500 + \left({}^{F_C}\!/_{F_T} \right) * 500 \qquad \textbf{(11)}$$

4.2.3 Representation and Schedule

A fairly large stack filter neighborhood was chosen. The stack filter inputs are selected from a 7 by 7 square neighborhood, which means the stack filter can have up to 49 distinct inputs. EA have been used to find optimal stack filters by a number of authors. A commonly used representation was suggested by Chu in [63]. This representation encodes the entire space of PBFs up to a given input size. It is impractical to use Chu's representation due to the size of the stack filters considered.

To reduce computation times the maximum number of minterms is restricted to 10 and the EA optimizes a set of 10 or less minterms for solving the problem. To implement this a fixed length string representation was used. A chromosome is made up of ten genes. Each gene defines a particular minterm through a bit-string, in our case 49 bits long. A NOP gene is included to include stack filters with less than 10 minterms, e.g.

```
[Gene 1]  MinTerm[0101000001010101000101010110011000001001011000001]
[Gene 2]  MinTerm[1000000010010001010101000001100100001100000010000]
[Gene 3]  NOP[]
[Gene 4]  MinTerm[1000000010010001010101000001100100001100000010000]
[    to   ]  .
[Gene 9]  NOP[]
[Gene 10] MinTerm[01000000000000000000000011000000000000011100000001]
```

This representation reduces the search space significantly to approximately $10*2^{49+1}$, however this representation does contain redundancy and the precise number of unique stack filters that this corresponds to is difficult to determine.

The representation does not enforce rotationally invariant stack filters. Morphological algorithms have achieved rotational invariance with non-isotropic structuring elements by using multiple, rotated versions of the structuring element. It is believed a similar approach can be used for the stack filter by adding additional, rotated min-terms.

The schedule for the EA is summarized in Table 6. Single point crossover was used. There are 9 valid crossover points corresponding to gene boundaries in the 10-gene chromosome. Bit-flip mutations are applied randomly to the 10-gene chromosome. A single application of the mutation operator can result in 1, 2 or 3 (randomly chosen) bits of the chromosome being flipped.

EA Parameter	**Number**
Population Size	500
Parents	80
Number of Generations	1000
Reproduction	
Elite	20
Mutated Elite	50
Crossover (parents chosen from top 60%)	250
Mutation (parents chosen from top 60%)	100
Random Generation	80

Table 6: EA Reproductive Schedule for Stack Filter Experiments

4.2.4 Experiment

Figure 53 depicts the training image and test images used. They were developed and used in [30]. Each texture has equal mean and variance and the boundary between textures is varied. To reduce EA execution times only a smaller sub-image, boxed in the training image of Figure 53 is used for calculating fitness. The training image contains four different textures. Examples of the binary truth masks are also illustrated in Figure 53 for clarity. Note, that a false or non-feature mask for each texture (not illustrated) is also defined so that only pixels defined as true or false contribute to the fitness and remaining pixels are treated as don't-care or unknown.

Two sets of stack filters, using the two different thresholding strategies described, are evolved. For each set, there are 12 stack filters optimized: a stack filter for each texture class at three different values of N (the number of iterations). Computation time restricted our choice of N to 1, 6 and 12. Note, these values do not guarantee convergence (or oscillation) although for N = 12, propagation of neighborhood samples can almost span a texture class.

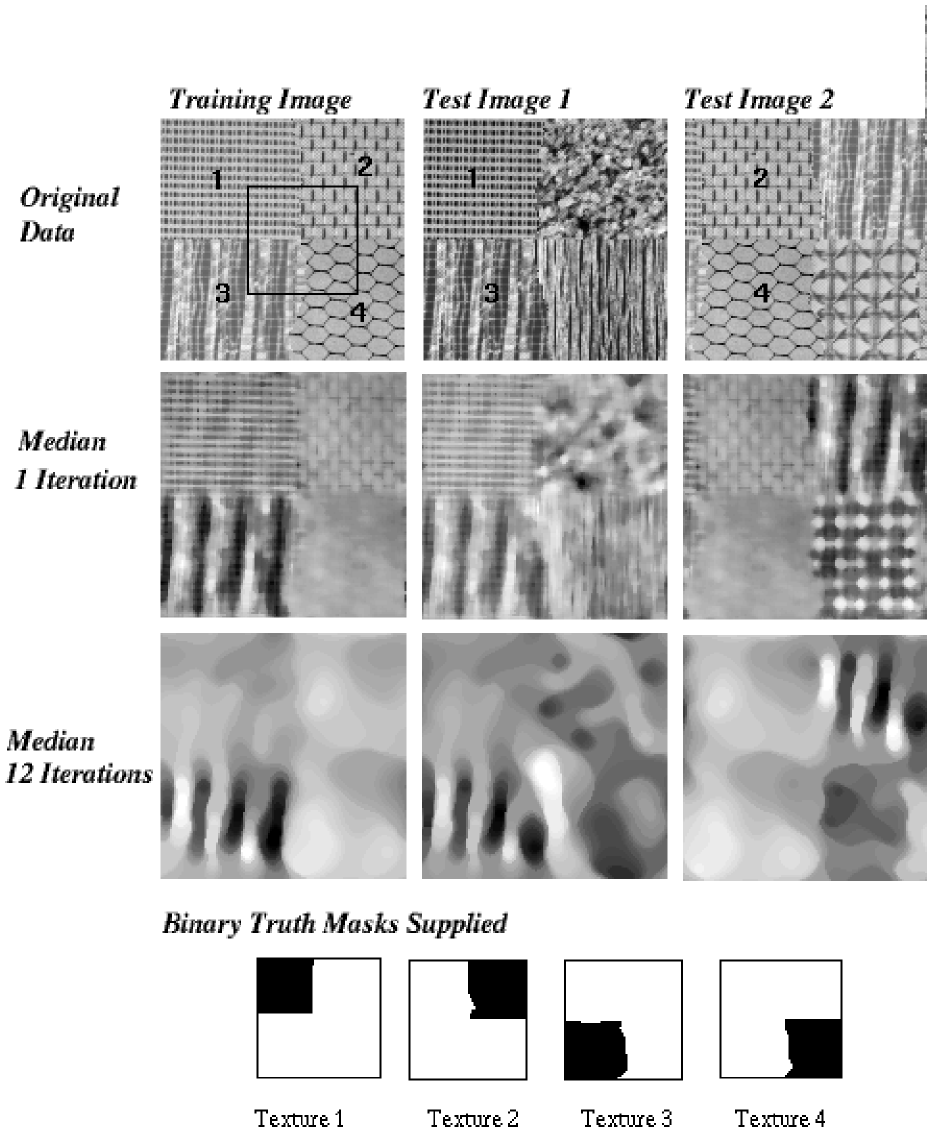


Figure 53: Training and Test Images

4.2.5 Results and Discussion

Table 7 reports the best fitness scores obtained by stack filters optimized using the different criteria. The fitness scores should only be used as a guide to the relative performance of the various techniques. Only a small training image was used (the smaller box in Figure 53) and therefore quantitative comparison using this Table can be misleading. By applying the optimized stack filters to the larger images in Figure 53 a more objective comparison can be made. The scores obtained on the larger images are presented graphically in Figure 54.

Texture Class	Invariant N=1	Invariant N = 6	Invariant N = 12	Median N=1	Median N=6	Median N=12
	Fitness out of 1000					
1 Top Left	635	701	670	804	906	945
2 Top Right	552	574	609	607	853	886
3 Bottom Left	617	651	595	703	909	916
4 Bottom Right	581	588	591	601	808	892

Table 7: Results on Training Images (Small training images)

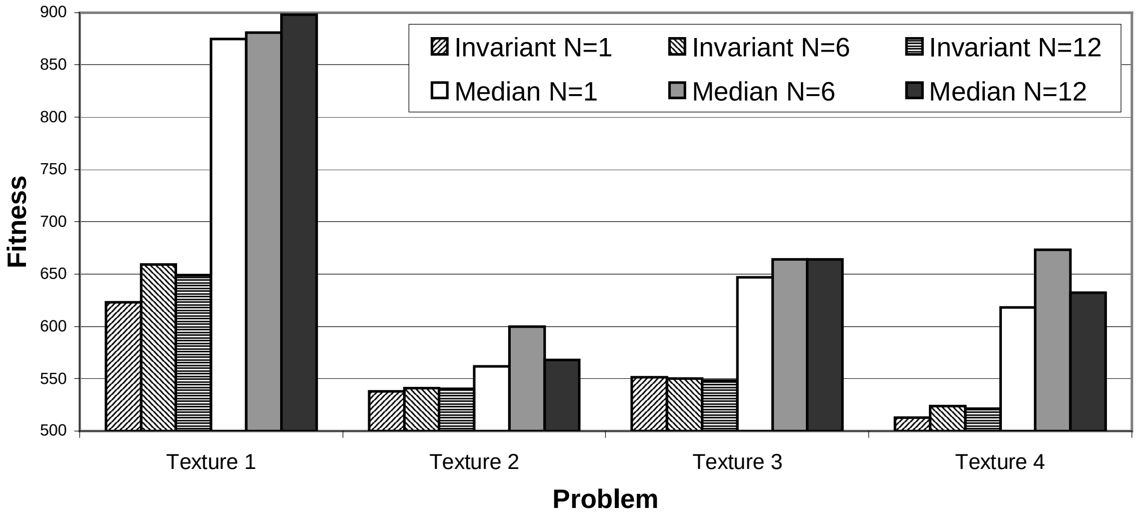


Figure 54: Results when filters are applied to larger images.

The best results on the larger images with the 'invariant threshold' strategy were for N=6, and these are shown in Figure 55, with the respective fitness scores. Either the EA was poorly suited to searching the space or more likely, due to the large number of runs performed, the invariant signal set of most stack filters does not characterize a texture very well.

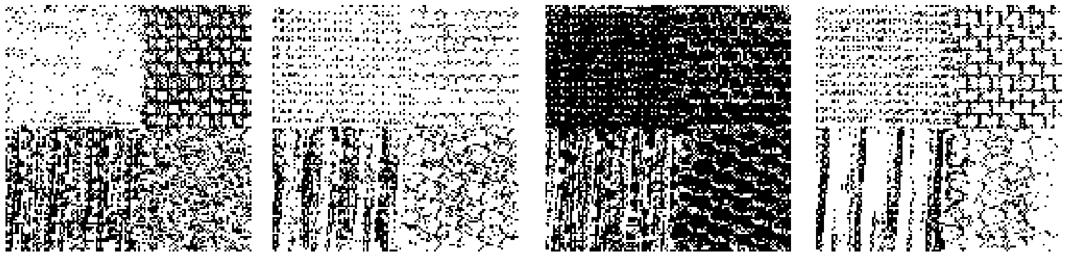


Figure 55: Output images from filters trained using Invariant Thresholding for N=6.

Fitness for texture 1 (left) : 659, texture 2 : 541, texture 3 : 550 and texture 4 (right) : 524

The performance of the 'median threshold' stack filters was varied. Significant performance improvement was observed as the number of iterations N, was increased. This is partly due to the level of smoothing which increases with N. While the N=12 filters achieved the highest scores on the training data, it can be seen by comparing Figure 56 and Figure 57, that the better result is not easy to discern, particularly for texture classes 2 and 3. This is a result of using one small training image. For N=12, stack filters were more dependent on the particular training data supplied since signals could propagate across the texture class and therefore the N=6 experiment is a better indicator of performance.

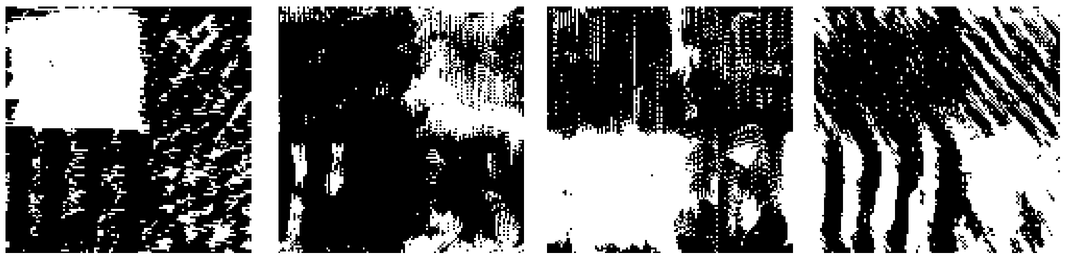


Figure 56: Output images from filters trained using Median Thresholding for N=6.

Fitness for texture 1 (left) : 881, texture 2 : 600, texture 3 : 664 and texture 4 (right) : 673

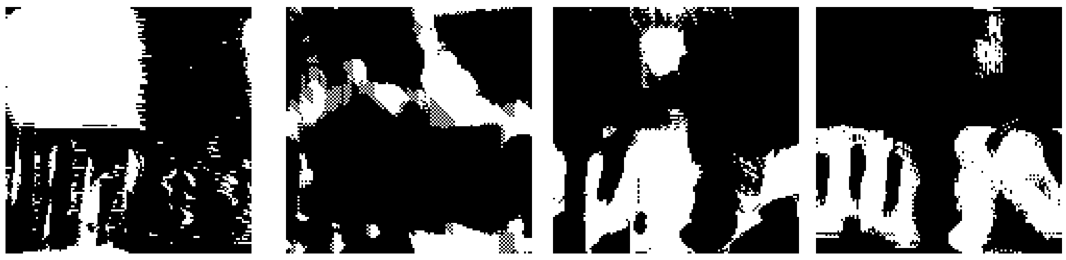


Figure 57: Output images from filters trained using Median Thresholding for N=12.

Fitness for texture 1 (left) : 898, texture 2 : 568, texture 3 : 664 and texture 4 (right) : 632

The best filters found using the median thresholding scheme with N=6 iterations were applied to the test images of Figure 53. The images of Figure 58, are in order of texture. That is, the output from the stack filter trained on texture 1 is on the left, through to the stack filter trained on texture 4 on the right. Note, filters trained on textures 1 and 3 were applied to Test Image 1 of Figure 53, and filters trained on textures 2 and 4 were applied to Test Image 2.

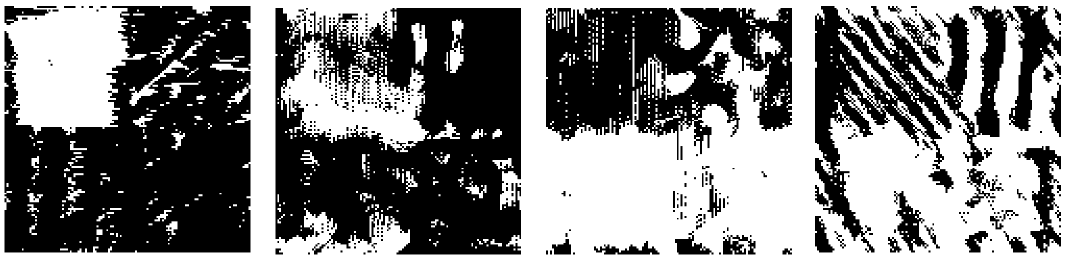


Figure 58: Output images from Median N=6 solution applied to test images.

Fitness for texture 1 (left) : 811, texture 2 : 579, texture 3 : 767 and texture 4 (right) : 605

In all cases, texture 1 was classified well but other textures such as 4 were difficult. It is suggested that the poor performance on texture 4 was due to the large areas of homogeneous gray-vales within the texture. Since the stack filter is applied as a shifting window filter it has difficulty producing output over these regions that is discernable from other homogenous areas such as those found in patterns 2 and 3. This is a result of the stack filter being based purely on relative information. While this property of stack filters is useful in terms of illumination insensitivity, a pathological problem to the stack filter architecture proposed is to classify a particular feature when supplied with target classification as data!

4.2.6 Summary

The advantage of the architectures considered lies in threshold decomposition and the stacking property. These lead to efficient hardware implementations, that could be potentially be implemented within maximally parallel cellular arrays. This experiment investigated the potential of such architectures for pattern recognition and compared architectural choices. By evolving stack filter neighborhoods, robust order statistic relationships within features of interest can be extracted. The limitations of this architecture were also demonstrated which is a result of level independence within the threshold decomposition architecture. Extension of these architectures is discussed in Chapter 5 with respect to Morphological Networks and Generalized Stack Filters.

4.3 Chapter Summary

This chapter has described experiments with maximally parallel cellular array chromosomes. Solution of practical AFE problems with these architectures is difficult with current FPGA devices, which limit the complexity of pixel processors. Two experiments, of more theoretical interest were therefore explored. Future chapters, which attempt to solve more practical problems, implement chromosomes with image pipelines. However, there are several conclusions from this chapter's experiments that are relevant to the rest of the thesis:

Evolving Cellular Automata

- Algorithm structure can be used to dictate a smaller, more relevant search space compared to the raw FPGA bit-stream.
- Conversely, FPGA resources can lead to algorithmic variants that include the relevant search space, but can also be implemented efficiently on FPGAs. An example of this is the *hybrid* XC6216 CA model.
- Implementing algorithms in hardware can also lead to a fine grain decomposition of the search space. This is useful for introducing problem specific constraints.

Evolving Stack Filters

- The flexibility of EA optimization means novel architectures and behavior can be investigated. Examples of this are the novel thresholding schemes used to train stack filters: *Invariance Thresholding* and *Median Thresholding*.
- Rank order relationships are useful in characterizing images, but the inability to discriminate differences in absolute gray values (level independence) is limiting. This problem is addressed in the next chapter with Morphological Networks.

Chapter 5

Evolving Network Architectures

Experiments of this chapter provide the framework, and point of departure for subsequent chapters that address practical AFE problems in multi-spectral data sets. Due to the difficulty in extending maximally parallel implementations to practical problems, chromosomes in this chapter are implemented with image pipelines. Section 5.1 describes image pipeline fitness evaluation on the Firebird CC in more detail.

Network architectures have many properties that are desirable in hardware implementations. They have also been used extensively for classification in AFE problems. This is particularly true for Neural Networks, introduced in the Literature Review. A variant of Neural Networks known as Morphological Networks are introduced. They are of particular interest to this thesis since they can be more efficiently implemented on FPGAs than Neural Networks and have a strong relationship to Mathematical Morphology. They also can be considered an extension of the Stack Filter architectures explored in Chapter 4 that include absolute gray value information. Both Neural Networks and Morphological Networks are described in more detail in Section 5.2.

Implementation of these networks using CC is described in Section 5.3. How EA can be used to optimize networks for particular problems is discussed in Section 5.4. Sections that follow evaluate and compare Neural and Morphological Network architectures with practical AFE problems.

5.1 Image Pipeline Fitness Evaluation

In this chapter, and in fact the rest of this thesis, chromosomes are implemented with image pipelines. This section describes the pipeline fitness evaluator that is implemented on the Firebird CC. Within the context of EA chromosomes, a pipeline has several advantages over maximally parallel implementations for solving practical problems:

- More complex algorithms can be implemented: To implement arrays of useful size, the pixel processor in maximally parallel cellular arrays is resource limited. In a pipeline, only 1 pixel processor needs to be implemented, and therefore much more complex processing is possible.

- Flexibility in image size: A pipeline implementation does not require a particular image size. Larger images simply result in longer execution times. This flexibility is essential in dealing with variable sized training images and data sets.

- Localized calculation of Fitness Metric: Calculations based on the entire image are more easily implemented in the pipeline architecture than maximally parallel implementations. The output of the pipeline is an image stream. This can be compared to a target classification stream using a small, localized fitness metric.

Implementations from this point on use the Firebird CC based around a Virtex 2000E FPGA. Figure 59 is a schematic diagram of the major components of the pipeline fitness evaluator using the Firebird CC. The *Generalized Chromosome* implements the pipeline to be evolved. It is configured through on-chip configuration registers. The *Generalized Chromosome* receives input data from one memory, performs the particular processing dictated by the configuration registers and then outputs the result to a second memory.

The *MemCounter* unit of Figure 59 implements a 20-bit counter, which is used to address both input and output memories. This unit also passes control information to the *Fitness-Metric* unit so that fitness is calculated only over valid output data. The entire design, including *Generalized Chromosome* and infrastructure, is controlled through an input control register, which contains a *data-count* register and a 1-bit *Pipeline-Reset*. If the *Pipeline-Reset,* in the top left of Figure 59, is set high, the pipeline and counters reset and the host has exclusive access to input and output memories. When the host sets the *Pipeline-Reset* low the memories are assigned to the *Generalized Chromosome* and the data is piped through the design. The *MemCounter* unit stops when it reaches the value stored in the *data-count* register and sets the *Pipeline-Done* output control bit.

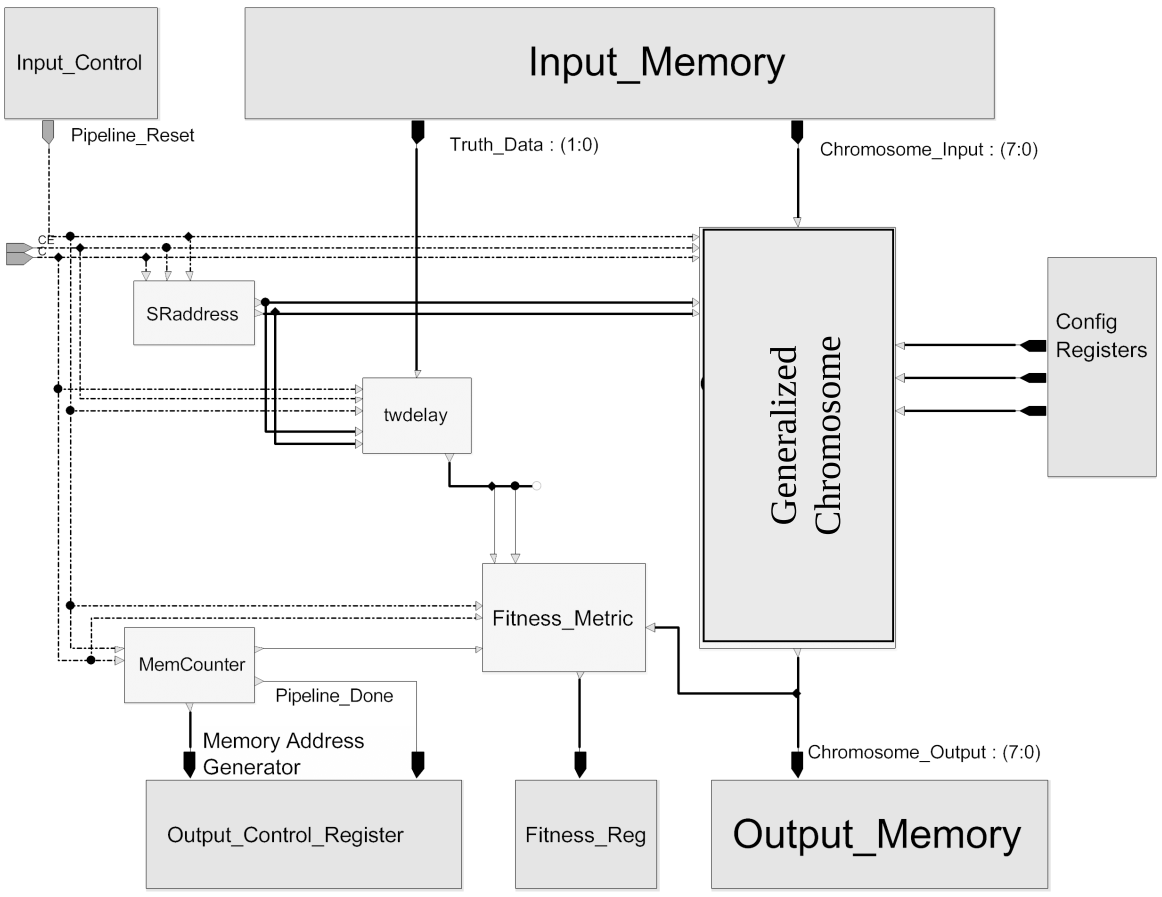


Figure 59: Pipeline Fitness Evaluator Architecture

At the same time data is piped through the *Generalized Chromosome,* the target classifciation (the true and false training data) is passed to a delay unit. This is called the *twdelay* unit in Figure 59. This unit implements a latency equivalent to the *Generalized Chromosome.* The latency adjusted truth data is then compared to the

Generalized Chromosome output in the *Fitness-Metric* unit. The fitness is then stored in the on-chip *fitness-register* where it is accessed by the host.

The fitness metric implements the weighted hamming distance introduced in Chapter 4. It is repeated in Equation 12.

$$Fitness = \left({}^{T_C}\!/_{T_T} \right) * 500 + \left({}^{F_C}\!/_{F_T} \right) * 500 \qquad \textbf{(12)}$$

T_C is the number of true pixels correctly classified by the network and T_T is the total number of true pixels in the training set. Similarly, F_C is the total number of false pixels correctly classified and F_T is the total number of false pixels in the training set. It is often convienient to represent this fitness score in terms of *detection rates* and *false alarm rates*. The detection rate is the percentage of feature pixels that are correctly classified as feature. The false alarm rate is the percentage of non-feature pixels that are incorrectly classified as feature. Equation 13 is another way of expressing Equation 12, where DR is the detection rate and FA is the false alarm rate.

$$Fitness = DR * 500 + (1 - FA) * 500 \qquad \textbf{(13)}$$

Note, this fitness measure is suitable only for binary or two-class classification problems. Secondly, only classification error is considered. No measure is made of the certainty in decision such as a distance from the decision boundary. The benefit of the weighted hamming metric is its simplicity of implementation in CC.

5.1.1 Host Program

The host program is responsible for implementing the evolutionary algorithm, writing and reading local memories on the CC as well as pipeline control. Figure 60 depicts the major components of the host program. Bold psuedo-code indicates where in the host program the major tasks are carried out.

At the top-level, the host program is primarily responsible for getting data to and from the CC local memories. At the start of the EA run, the host program writes training data to the Input Memory. At the end of the run, output images are read from the Output Memory. The top-level object also instantiates a *Population* object and controls the number of evolutionary iterations or generations.

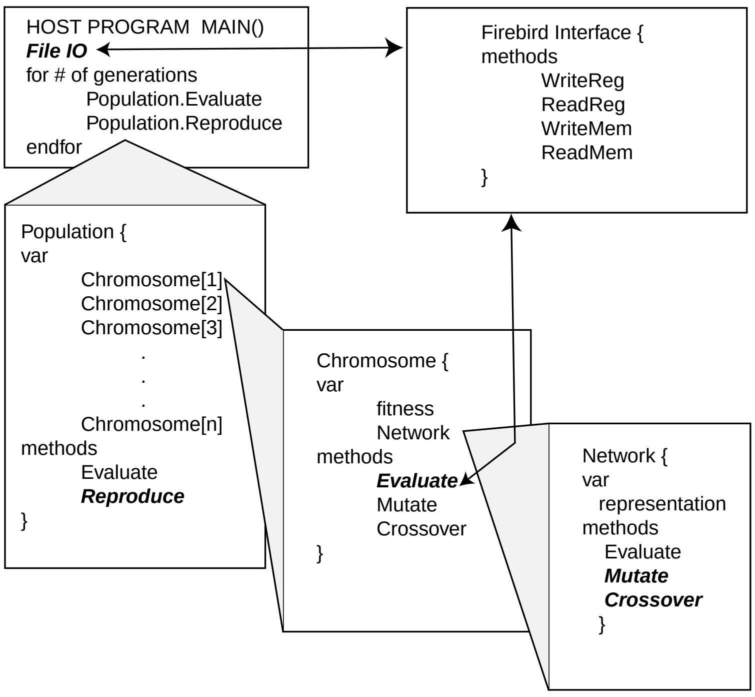


Figure 60: Overview of Host Program

The *Population* object instantiates a population of *Chromosome* objects, each representing a candidate solution. The *Population* object implements the genetic selection and reproduction algorithms.

The *Chromosome* object does not explicitly contain the chromosome for a particular candidate solution. Instead, the *Chromosome* objects instantiate another problem specific object, which contains the actual chromosome. In this chapter we are interested in evolving the network architectures and therefore Figure 60 includes the additional *Network* object. The primary role of the *Chromosome* object is to instantiate and evaluate a particular *Network* on CC. To do this, the *Chromosome* object requests the hardware configuration registers from the Network object. It then writes this information to the *Generalized Chromosome* configuration registers. The *Chromosome* object is also responsible for initiating the pipeline and retrieving the fitness score.

The *Network* object contains the problem specific representation and is primary role is to implement genetic operators such as crossover and mutation. The representation used to apply genetic operators is often significantly different from the corresponding hardware configuration registers. This is discussed more in Section 5.4. The *Network* object is therefore required to translate the internal representation to the relevant hardware configuration so it can be configured on the CC by the *Chromosome* object.

5.2 Network Architectures

Network architectures consist of a pipelined array of individual nodes. Network architectures have several properties that lead to efficient hardware implementation. These include:

- Inherent Parallel Processing: The final output of a network is a result of partial calculations performed by each node.
- Simple Processing Elements: Each node of the network need only be capable of solving part of a particular problem and therefore are relatively simple
- Modular: Nodes are usually homogeneous across the network leading to simple large-scale designs.

For these reasons, networks appear to be a good starting point from which to develop hardware efficient *Generalized Chromosomes* for AFE problems. This is not a new thing, and is partly why Neural Networks have received considerable attention for solving the classification aspect of AFE problems. They have also been applied to multi-spectral classification by several authors [114], [132].

Neural Networks have been implemented on FPGAs by several researchers to accelerate both training and application of particular networks. A fundamental operation in neural networks is multiplication. This can be expensive to implement on FPGAs as the number of nodes and connectivity within the network grows. Several techniques have been used to reduce this problem: implementation of partially connected neural networks [133], and time multiplexing of network nodes using partial reconfiguration [134]. FPGA implementations can provide significant speed-up compared to software implementations, and have the advantage of flexibility, which is of benefit to many applications. However, for other applications FPGAs cannot provide sufficient densities of neurons and ASIC implementations, often analogue, are the preferred solution.

Morphological Networks have much in common with Neural Networks but represent a fundamentally different approach. They have been shown to have equivalent classification power to neural networks [135] and can be implemented on FPGAs much more efficiently than traditional neural networks.

Traditional neural networks, using linear perceptrons, involve multiplication of inputs by weights, and then summing the results. This linear operation is then followed by a non-linear thresholding operation to produce the perceptron output. This is illustrated in Figure 61a. In the morphological case, the operations of multiplication and addition are replaced by addition and maximum / minimum respectively. The morphological perceptron is illustrated in Figure 61b.

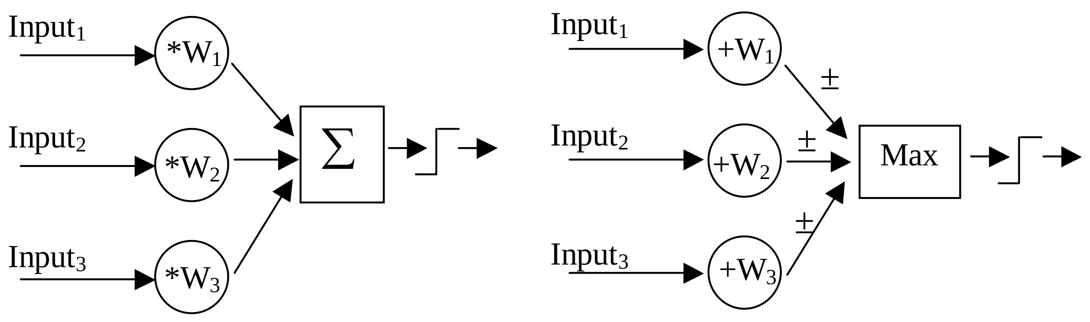


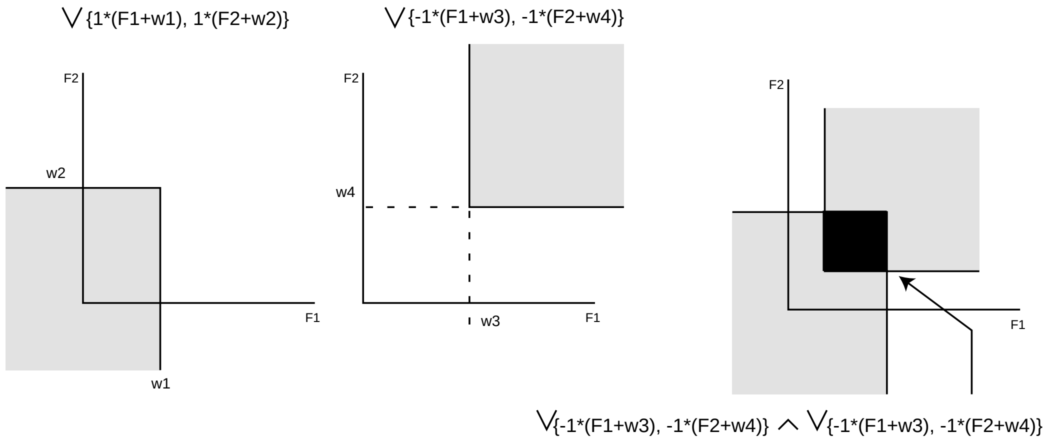
Figure 61: a) Linear Perceptron and b) Morphological Perceptron

The definition of the morphological perceptron presented comes from work presented in [136]. A multiplicative weight of ±1 is also associated with each input, which is described as an excitory / inhibitory weight.

Several other researchers have suggested morphological networks but in other forms. Morphological Networks presented in [137] use morphological maximum and minimums to replace the linear perceptron addition, but use multiplicative weights on the inputs. Min-Max classifiers in [138] are similar in principle to Figure 61b, but were only considered for the binary case and additive input weights were not used.

5.2.1 Learning Algorithms

Before describing how EA is used to optimize network architectures, it is worth considering how more traditional learning algorithms can be implemented. The classification power of morphological networks was investigated in [135]. A learning algorithm was presented that could find arbitrary decision surfaces with a 2-layer network.

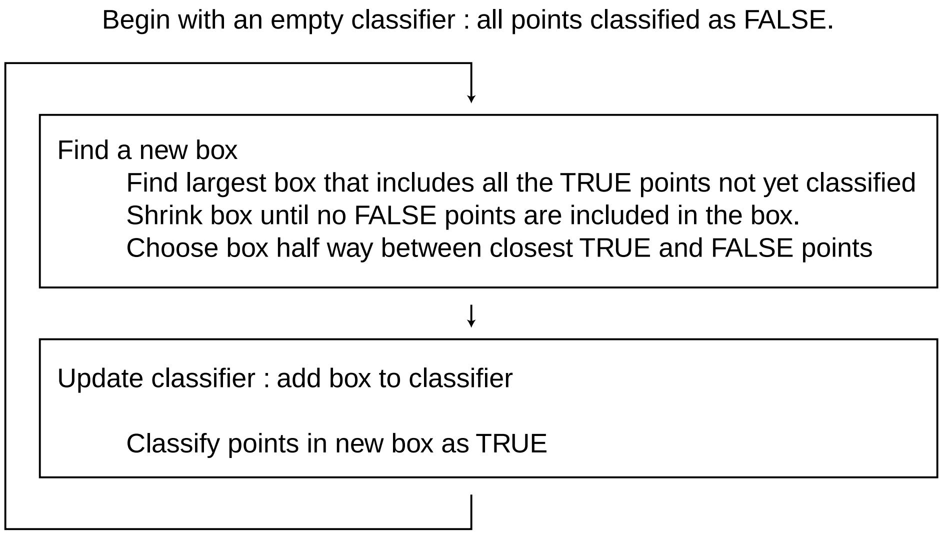


⋁{1*(F1+w1), 1*(F2+w2)}
F2
w2
F1
w1
⋁{-1*(F1+w3), -1*(F2+w4)}
F2
w4
w3
F1
F2
F1
⋁{-1*(F1+w3), -1*(F2+w4)} ∧ ⋁{-1*(F1+w3), -1*(F2+w4)}

Figure 62: Morphological Perceptron and Feature Space

This learning algorithm is most easily understood by looking at decision surfaces formed by morphological perceptrons in feature space. The two illustrations on the left in Figure 62 illustrate these decision surfaces. The perceptron is defined as the maximum of the set of weighted inputs. After thresholding at 0, this divides the feature space into two regions. It also leads to a decision surface that is parallel to the feature space axis. Note that output from two perceptrons can be combined with a minimum (a two-layer network) to form a box. This is illustrated on the right of Figure 62.

The learning algorithm proposed by Sussner [135], is an iterative algorithm consisting of two steps. Pseudo-code for this algorithm is shown in Figure 63. With enough nodes, this algorithm can find a 2-layer morphological network to form arbitrary decision surfaces.



```
Begin with an empty classifier : all points classified as FALSE.

Find a new box
        Find largest box that includes all the TRUE points not yet classified
        Shrink box until no FALSE points are included in the box.
        Choose box half way between closest TRUE and FALSE points

Update classifier : add box to classifier

        Classify points in new box as TRUE
```

Figure 63: Learning Algorithm Pseudo-code

This learning algorithm was implemented and its application to feature space illustrated in Figure 64. On the left of Figure 64 is the training data where gray points represent true and black points represent false for a two-class problem. In the result image, on the right of Figure 64, the black region corresponds to points the morphological network classifies as true.

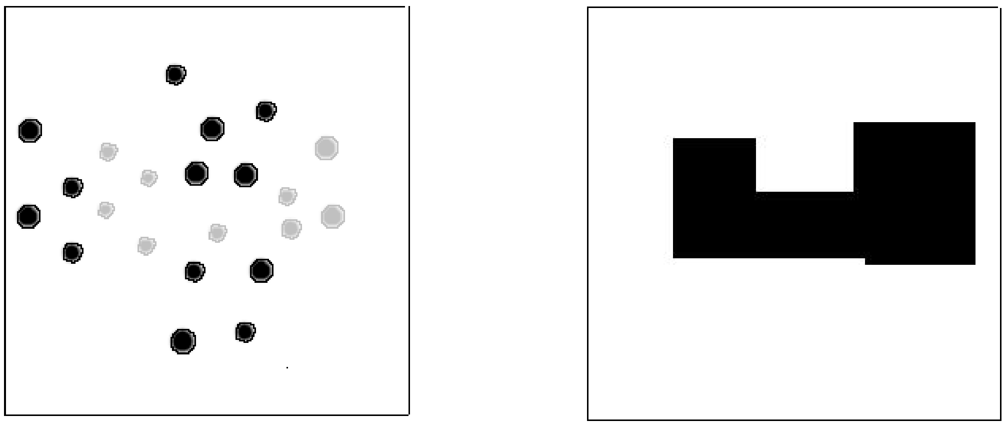


Figure 64: Morphological Networks Applied to Feature Space (a) Training Data (b) Result

The learning algorithm presented in [135] is capable of finding a solution to arbitrary two-class problems. However, the number of nodes required depends on the problem, and in worst case, may be equal to the number of training points. To find learning algorithms that can optimize a finite number of nodes, and accept classification error is a more difficult problem.

A similar problem is faced in the field of neural networks. Back Propagation can be applied to fixed sized networks, but is often caught in local minima. Also, Back Propagation type learning algorithms require the neural network activation function to be differentiable. It is therefore common in Neural Networks to use Sigmoid activation functions. These can also be expensive to implement on FPGAs.

When EA is used to optimize a neural network the differentiable activation function is no longer required, and a hard threshold at zero can be used. This threshold is easily implemented on FPGAs. An interesting application of EA to this problem was reported in [139]. This work describes using Genetic Programming to *discover* learning algorithms for neural networks with step activation functions, rather than optimizing weights directly.

5.3 Implementation

This section describes implementation of morphological and neural networks in CC. The circuits to be described are implemented within the *Generalized Chromosome* component of the pipeline fitness evaluator architecture described in Section 5.1. The hardware resources required to implement a Morphological and Neural Network of similar size are compared in Section 5.3.4.

5.3.1 The Morphological Perceptron

Morphological Networks (MN) are of particular interest since multiplication is avoided. Figure 65 illustrates the input weight circuit for the MN. A 9-bit 2's complement adder is used to sum an 8-bit signed input value with an 8-bit signed weight. The final output of the morphological perceptron is a thresholded Minimum, or Maximum, of the weighted inputs. Therefore, only the MSB of the adder output is required in subsequent processing. This bit indicates whether the weighted input is above or below the threshold. The MSB is conditionally inverted using a 2-input LUT (look-up-table) according to the ±1 multiplicative weight associated with the input.

The register (FDRE) is used to pipeline the network so that high clock frequencies can be maintained.

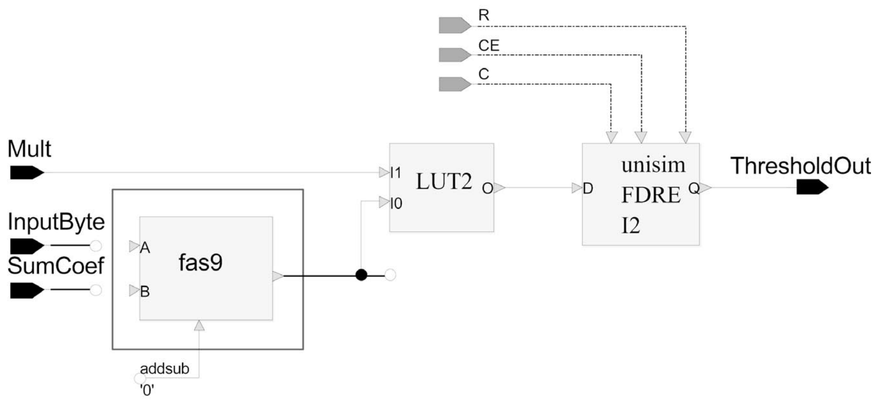


Figure 65: Morphological Weight Circuit

A 12 input perceptron was implemented and therefore 12 weight circuits are used. Since output from the Morphological weight circuit is binary, only 6 4-input LUTs are required to produce the perceptron output. These LUTs each take a control line that dictates whether they implement a maximum (OR) or minimum (AND) of the 12 inputs.

5.3.2 The Linear Perceptron

In contrast, Figure 66 illustrates the input weight circuit for the multiplicative neural network. A pipelined multiplier is used (Mult3by8 and compl units) which is comparable in size to 4 9-bit adders. In the linear perceptron, the output of each weight circuit must be summed with other weighted inputs before thresholding can occur. To maintain full precision the input weight circuit needs to output a 12-bit result. The 12 weight circuit outputs are then combined with a binary tree of adders and produce a 16-bit result. The 16-bit result is then thresholded at 0 to produce the perceptrons single bit output.

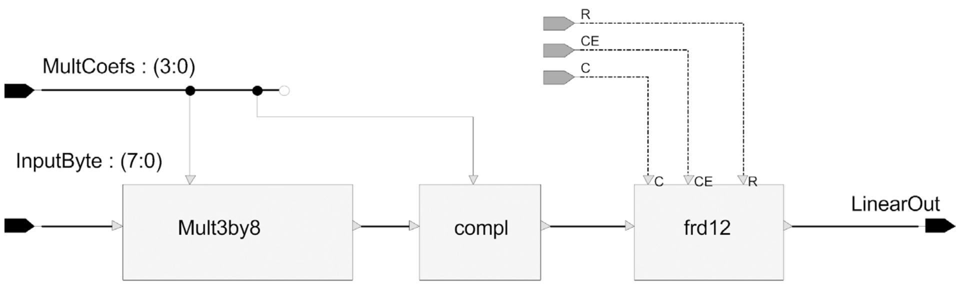


Figure 66: Neural Network Weight Circuit

5.3.3 2-layer Networks

Two-layer networks are built from the morphological and linear perceptrons. Each network has 12 inputs, 4 hidden layer nodes, and a single output node. This is depicted in Figure 67 for clarity.

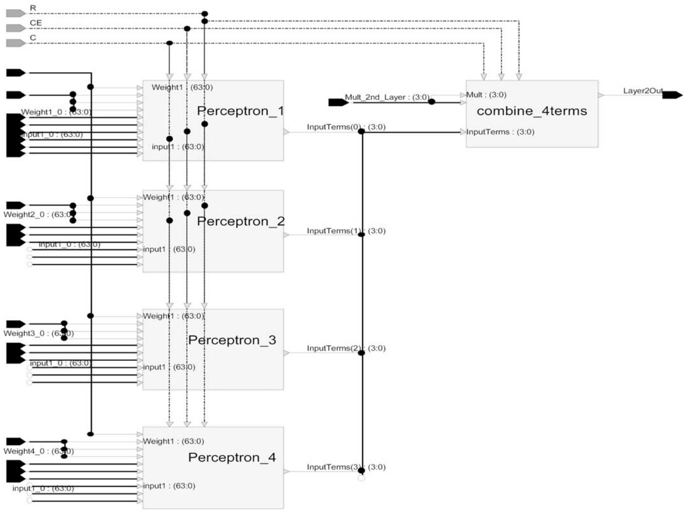


Figure 67: The 24-input, 4-hidden, 1-output network

The circuit used to implement the output node (*combine_4terms*), is illustrated in Figure 68. 2 input LUTS are used to invert the inputs from the previous layer according to ±1 multiplicative weights. These 4 conditionally inverted inputs are then passed to 4-input LUTs that can be configured to implement either AND or OR. In this experiment the conditionally inverted perceptron outputs are combined with a logical AND. This is a common way to combine hyper-planes for neural networks, and also in morphological networks, where it is considered a minimum of the first layer maximums.

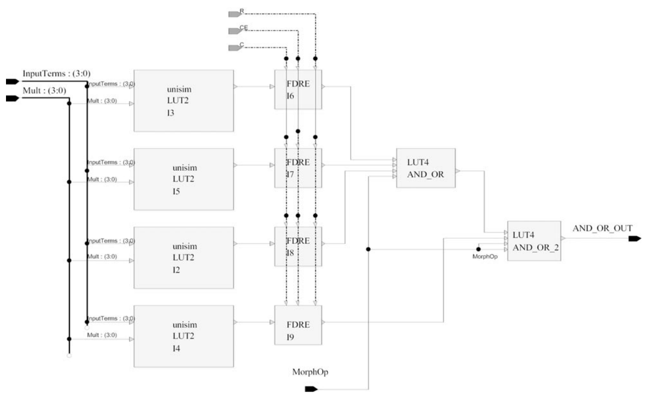


Figure 68: The AND/OR Output Node

5.3.4 Hardware Resource Comparison

Table 8 summarizes the resource requirements for the morphological and neural network implementations. There is a resource overhead when using the Firebird CC for memory, clock and PCI bus interface circuits. An 'empty' design is therefore also implemented so that the relative cost of network architectures is more accurate.

Design	Virtex Logic Block utilization (slices)	Total Utilization (%)	Design Utilization (%)
Firebird Infrastructure	1935 out of 19200	10	0
Morphological Network	2432 out of 19200	12	2
Neural Network	3470 out of 19200	18	8

Table 8: Resource Usage Comparison

The right column of Table 8 illustrates the efficiency with which the morphological network can be implemented. The neural network is approximately 4 times larger than the morphological design. A timing constraint of 66MHz was easily met by the morphological design with a pipeline latency of 2 clock cycles. These constraints were also met in the neural network design, however 3 pipeline stages were required due to the linear perceptron adder network. Even with a 3-stage pipeline, several iterations of Place and Route were required. This suggests larger neural network implementations may require additional pipelining and therefore more resources.

5.4 Representation and Schedule

Chromosomes for evolving the networks are made up from the weights associated with each input. Table 9 summarizes the software Chromosome to which genetic operators are applied, and the corresponding on-chip configurations registers. In hardware, the neural network multiplicative weights are 4-bit signed integers corresponding to the range –7 to 7. This representation was also used in the software chromosome.

	Morphological Network			**Neural Network**		
	Name	*Software Chromosome*	*Registers In CC*	*Name*	*Software Chromosome*	*Registers In CC*
Layer-1 Inputs (ith input)	Coef[i]	0 to 7	$Mult[i]*\left(2^{Coef[i]}-1\right)+Offset[i]$	Mult[i]	-7 to 7	-7 to 7
	Offset[i]	-32 to 32				
	Mult[i]	-1 or 1				
Layer-2 Inputs (ith input)	Mult[i]	0 or 1	Inverter	Mult[i]	0 or 1	Inverter

Table 9: Chromosome and on-chip registers for Morphological and Neural Networks

In the morphological case, the weights in hardware are 8-bit signed two's complement integers. To reduce the search space for the morphological network a more complex software representation was used. At the beginning of the run, the *offset* values associated with each input are set to zero. Therefore, only the *Mult* and *Coef* components in Table 9 are used and weights are restricted to powers of two. After a pre-defined number of generations, the *Offset* component is then allowed to mutate. By initially restricting weights to powers of two a broader ranger of network weights can be evaluated. When the best networks under this criterion have been found, they can then be fine-tuned through mutation of the *Offset* component. In the second layer, the chromosome is equivalent for both morphological and neural networks. This means the search space size is approximately the same, which is useful for making comparisons.

A generational EA with elitism is used. The reproductive schedule is summarized in Table 10. The CC evaluation time was approximately 120 seconds, although optimal networks were often found much earlier. A comparison to software training times is not made. This chapter is primarily concerned with the classification accuracy of the neural and morphological networks. Detailed comparison of CC and software training times for network architectures is presented in Chapter 7.

EA Parameter	**Number**
Population Size	200
Parents	80
Number of Generations	500
Reproduction	
Elite	20
Mutated Elite	50
Crossover (parents chosen from top 60%)	250
Mutation (parents chosen from top 60%)	100
Random Generation	80

Table 10: Reproductive schedule for morphological and neural network experiments

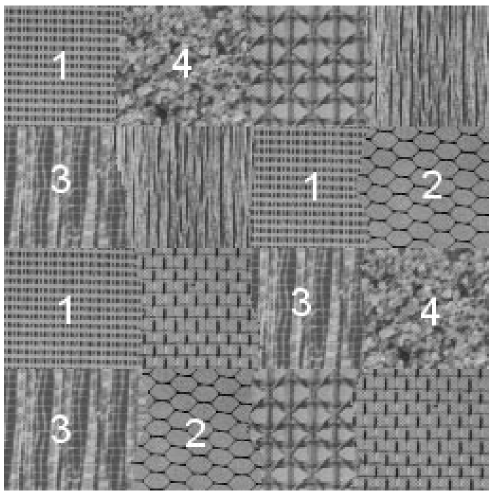
A single point crossover is used and was applied at perceptron boundaries. Two parents are selected, and a random number of perceptrons (1,2 or 3) are swapped. Mutation is applied in either 1 or 2 randomly chosen weights.

5.5 Experiments with Texture Classification

In this section, the classification accuracy of the morphological and neural network implementations is compared. The problem is similar to the texture classification task of the previous chapter. The speed of the CC fitness evaluation means a larger training image can be used for the texture classification problem, than was possible in Chapter 4. This larger image can be seen in Figure 69. Four patterns were chosen, indicated by numbers 1 through 4, and therefore 4 networks of each type (Morphological and Neural) were optimized.

In this experiment, a more conventional approach is taken and the texture classification task is decomposed into feature extraction followed by classification. Feature extraction is performed in software. Twelve features are generated and downloaded to the Firebird Local Memory. Each image pixel is therefore characterized by a 12-element vector in feature space. Classification is then carried out in CC where both morphological and neural networks are applied.

The twelve features are generated by calculating a pattern spectrum [34]. This was described in detail in the Literature Review. In short, morphological opening and closing is applied to the texture image of Figure 69a using successively larger structuring elements. Figure 69b depicts 6 of the 12 features produced. The left column shows the pattern spectrum images after opening at 3 scales using a square-structuring element of size 3,7 and 11. The right column shows the pattern spectrum images produced from closing using the same sized structuring elements. Note, images from structuring elements of size 5, 9 and 13 are not shown.

A) 

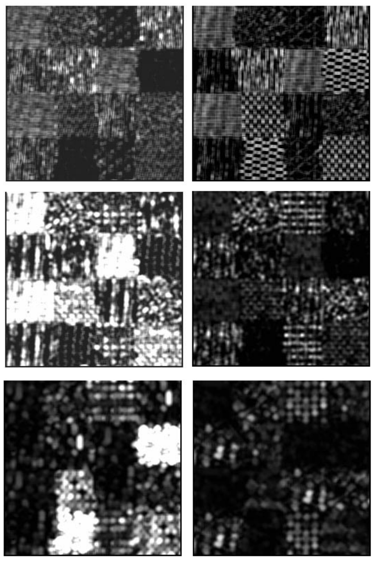
B) 

Figure 69: A) The Original Image for Texture Classification. 1: Check, 2: Honey Comb, 3:Snake skin and 4: Rubble. B) 6 of the 12 Pattern Spectrum features. Left column: pattern spectrum after opening with structuring elements of size 3, 7 and 11. Right column: pattern spectrum after closing with structuring elements of size 3, 7 and 11.

5.5.1 Results and Discussion

Training data was only supplied for the left half of the texture image in Figure 69. Both morphological and neural networks were evolved in 5 independent runs for each texture. The results of these experiments are reported in Table 11. Numbers represent classification accuracy, calculated using the hamming metric. A perfect classification results in a score of 1000. The software experiment used the back propagation algorithm. Multiple runs of the algorithm were used, but were often caught in local optima. The reported results were the best score obtained from the multiple runs. There are many other neural network learning algorithms that could find global optima more consistently, however they were not investigated.

Morphological Network						Mean	SD	Neural Network Back Propagation
1: Check	950	937	917	945	942	938	12.7	846
2: Honey	969	965	962	968	955	964	5.6	871
3: Snake	846	841	839	837	841	841	3.3	734
4: Rubble	810	793	807	801	813	805	7.9	657
Neural Network								
1: Check	976	975	976	975	976	976	0.5	
2: Honey	989	988	987	987	986	987	1.1	
3: Snake	873	873	877	879	882	877	3.9	
4: Rubble	855	847	846	855	858	852	5.4	

Table 11: Training Scores for Morphological and Neural Networks

The optimized networks of Table 11 were then applied to the right side of Figure 69, and fitness scores calculated to measure out-of-sample performance. The test scores are summarized in Table 12. It is observed that out-of sample results are very similar for both morphological and neural network cases with respect to training data scores.

Morphological Network						Mean	SD	Neural Network Back Propagation
1: Check	952	956	919	938	958	945	16.3	855
2: Honey	947	939	944	954	932	943	8.3	831
3: Snake	781	770	771	767	775	773	5.4	645
4: Rubble	739	735	736	737	744	738	3.6	623
Neural Network								
1: Check	983	979	981	981	980	981	1.5	
2: Honey	973	977	977	979	970	975	3.6	
3: Snake	782	787	778	796	791	787	7.1	
4: Rubble	792	803	787	797	803	796	7.0	

Table 12: Test Scores for Morphological and Neural Networks

The output images for both the morphological and neural networks (on the complete image) are illustrated in Figure 70. In absolute terms, better accuracy has been reported by researchers dedicated to texture characterization [30]. However, this experiment is primarily concerned with the relative performance of the network architectures. Both networks found better solutions to those found in Chapter 4. Also, the neural network produces a cleaner output image than the morphological network. This indicates that the neural network is potentially making more useful decision surfaces than the morphological network for this problem. This is explored further by applying both networks to multi-spectral AFE in the next section.

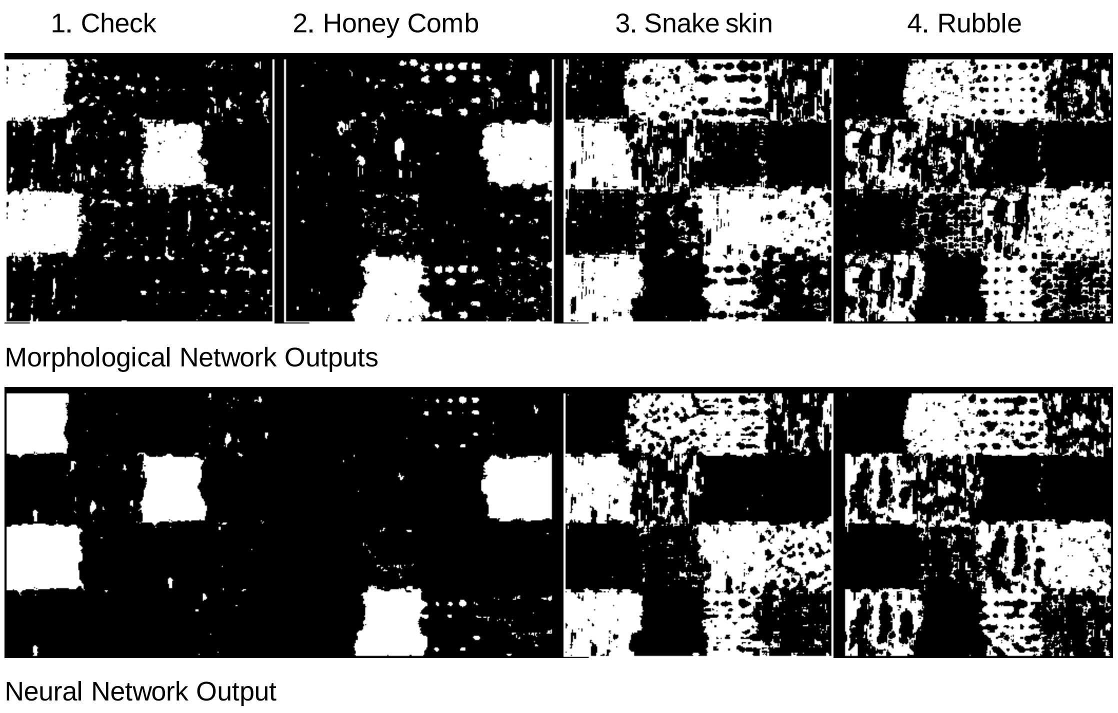


Figure 70: Output images on Training Data

5.6 Experiments with Multi-Spectral AFE

In multi-spectral data processing, a D element vector in spectral space characterizes each pixel, where D is the number of spectral channels in the image cube. In most traditional approaches to multi-spectral image processing classification is applied directly to this spectral space. In this section, the morphological and neural networks are applied in this way.

A 10-band multi-spectral data set was used. Two of the twelve inputs associated with the morphological and neural networks are therefore not used. The training images are depicted in Figure 71, together with the target classifications. White indicates the feature of interest, gray indicates non-feature and black corresponds to *don't know* and does not contribute to the score. The training data in Figure 71 was generated with a graphical *point and paint* program and can appear arbitrary. Regions that are ambiguous are left as *don’t know* to avoid providing inconsistent training data.

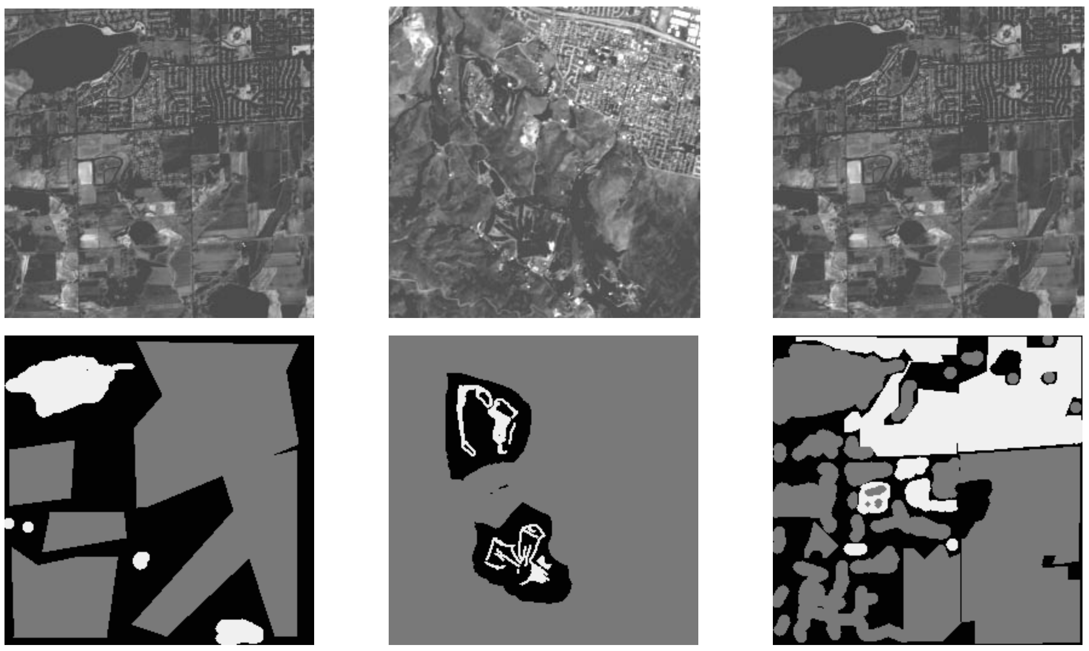


Figure 71: Multi-Spectral Training Set A) Water, B) Golf Courses and C) Urban Areas

The problem set was designed to span a range of difficulties. On the left, the feature of interest is water. This is the easiest problem of the three since water has a unique spectral signature. The second problem is to identify the golf courses. It is believed that this problem is of moderate difficulty but should have distinguishable spectral properties. The type of grass used in golf courses is often unique and therefore may be detected with purely spectral information. The third training image specifies urban or 'built-up' areas as the feature of interest. Urban areas can include a wide variety of materials and therefore spectral signatures. It is believed this is the hardest problem to be solved with spectral information alone.

Both morphological and neural networks were evolved in 5 independent runs for each feature. Optimized networks are then applied to test images illustrated in Figure 72.

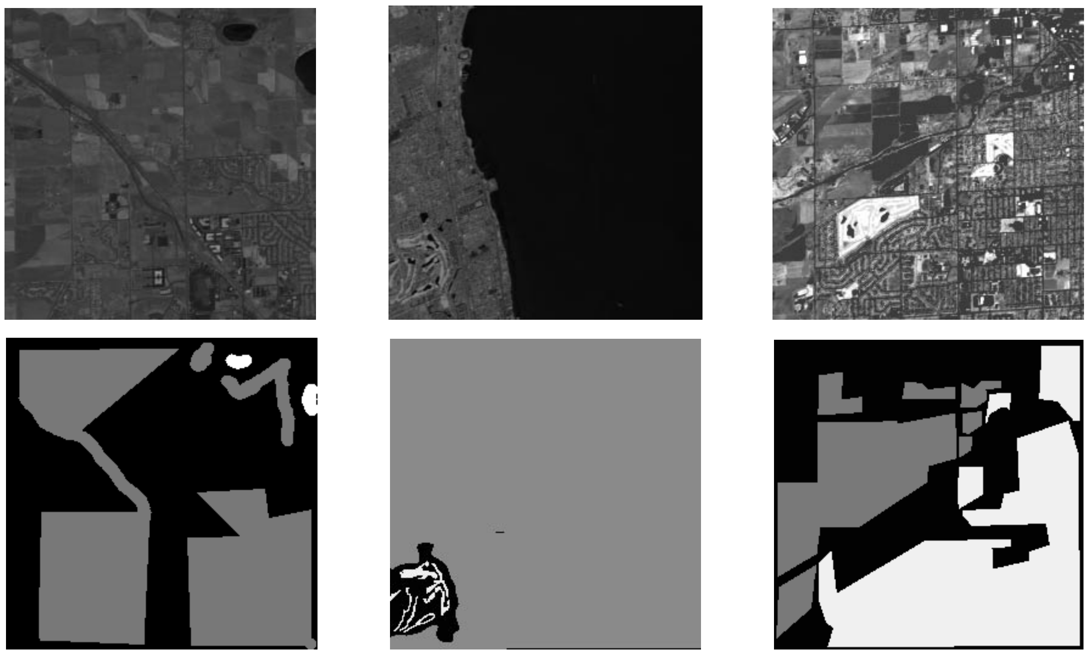


Figure 72: Multi-Spectral Test Set A) Water, B) Golf Courses and C) Urban Areas

5.6.1 Results and Discussion

The results on the training data are reported in Table 13. The optimized networks were then applied to the test images and results are reported in Table 14. Again, numbers represent classification accuracy based on the hamming fitness metric and a perfect classification is 1000.

Figure 73 shows the output images for the best morphological and neural networks found for each feature of interest. Figure 74 shows the output images from applying the networks to the test images.

Morphological Network						*Mean*	*S.D.*	*Neural Network Back Propagation*
Water	999	999	999	999	999	999	0.0	
Golf	955	957	948	958	953	954	3.73	
Urban	770	767	767	746	748	759	9.26	
Neural Network								
Water	999	999	999	999	999	999	0.09	999
Golf	963	960	964	962	962	962	1.65	937
Urban	792	780	791	781	798	788	7.70	756

Table 13: Fitness Scores obtained by Morphological and Neural Networks on training data.

Morphological Network						*Mean*	*S.D.*	*Neural Network Back Propagation*
Water	807	930	786	787	791	820	61.9	
Golf	894	826	885	832	891	857	33.6	
Urban	660	657	610	670	643	648	23.3	
Neural Network								
Water	810	922	889	833	841	859	45.5	864
Golf	933	899	919	853	877	896	32	768
Urban	738	690	692	671	698	698	24.6	683

Table 14: Fitness Scores for Morphological and Neural Networks applied to test images.

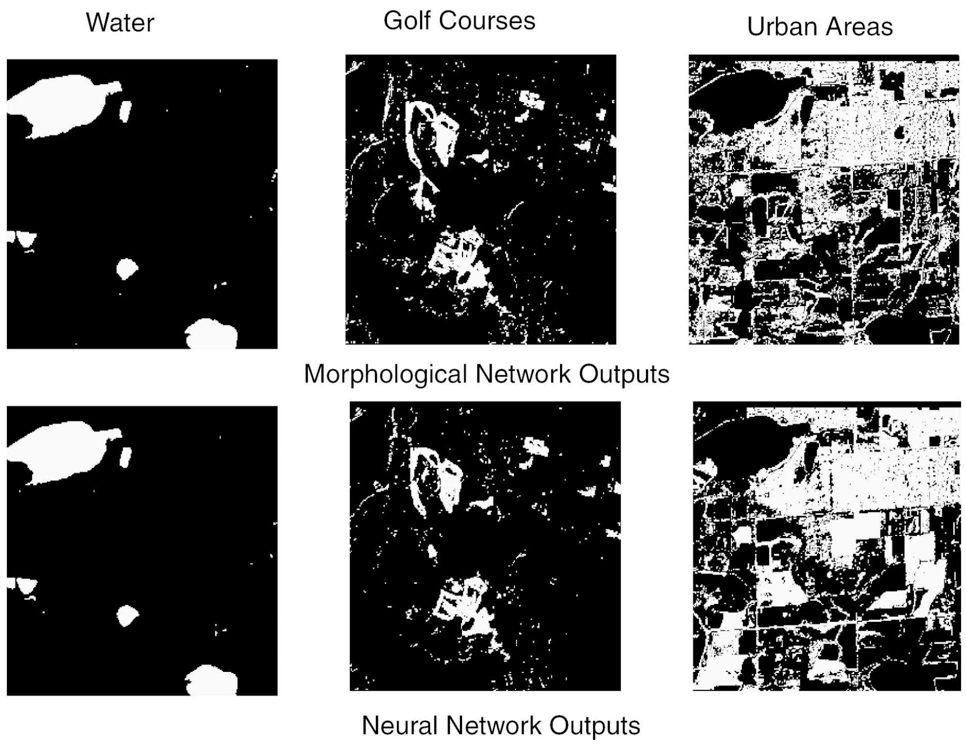


Figure 73: Output Images on Training Data

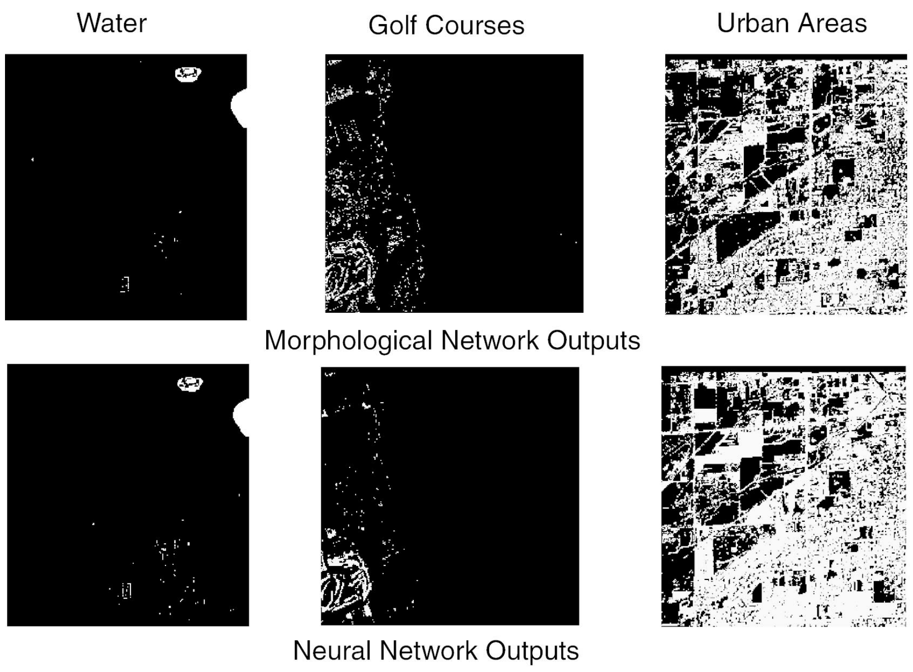


Figure 74: Output Images on Test Data

The neural network architecture out performs the morphological architecture in all problems. Both types of network found the multi-spectral feature identification problems progressively more difficult as expected. The neural network is seen to perform better on test images than the morphological network. However, this does appear consistent with the lower training data scores that were obtained by the morphological network. It is hypothesized that fine-tuning the Morphological Network *offset* values may lead to more brittle solutions, and therefore adversely affect generalization performance. To test this hypothesis, the Morphological Network was applied to the data set a second time. In this case, the *offset* values were kept at a constant of zero throughout evolution. The performance of the optimized network is summarized in Table 15 for the training images and Table 16 for the test images.

Morphological Network	Training					*Mean*	*S.D.*
Water	999	999	999	999	999	999	0.0
Golf	948	948	947	948	947	948	0.5
Urban	760	760	750	765	735	754	11.9

Table 15: Training Example Scores for Morphological Network with constant *offset* values.

Morphological Network	Testing					*Mean*	*S.D.*
Water	924	924	913	849	914	905	31.6
Golf	922	922	885	922	918	914	16.2
Urban	659	636	619	661	658	647	18.5

Table 16: Training Example Scores for Morphological Network with constant *offset* values.

By comparing Tables 15 and 16 to the Morphological Network results in Tables 13 and 14, it can be seen that although slightly better scores were achieved by fine-tuning the *offset* values, there was significant improvement in generalization by keeping these values constant. For this reason, offset values are not used in experiments involving morphological network components that are described in the next 2 chapters. The following conclusions are made:

1. The Neural Network architecture seems to have greater classification power than the morphological network architecture. However, it should be noted that the Neural Network used approximately 4 times the number of resources of the Morphological Network. An interesting direction, which was not investigated, would be to compare the Neural Network with a Morphological Network 4 times the size.
2. By confining Morphological Network coefficients to powers of 2, better generalization performance was observed.
3. Both spectral and spatial information are important in many multi-spectral feature identification problems of interest.
4. If a network is to be implemented in hardware, evolutionary search is a well-motivated learning algorithm.

5.7 Further Discussion

In this chapter, the neural network architecture was able to form more useful decision surfaces than the morphological architecture. In terms of hardware resources, this is not surprising since the neural network had approximately four times the computational resources of the morphological network. A topic of future work is to enhance the classification power of the morphological network. One way of extending the morphological network is inspired by stack filters, introduced in Chapter 4. Figure 75 highlights the difference in morphological network and stack filter architectures. If additive weights are applied to stack filter inputs, the two architectures appear very similar. For morphological networks on the right of Figure 75, the output from each minimum (perceptron) is thesholded. However, if the maxium of the minterms is found without thresholding, a PBF is formed. This can be seen on the left of Figure 75.

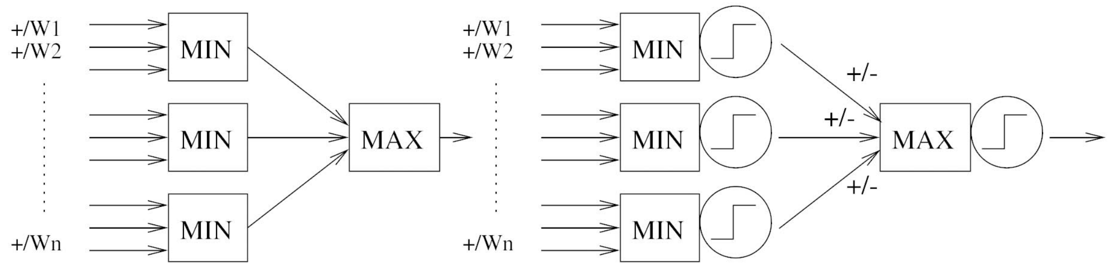


Figure 75: Left: Generalized Stack Filter and Right: Morphological Network

This architecture can be considered a subset of generalized stack filters (GSTFs) presented in [140]. GSTFs allow the PBF to receive input from more than one level in the threshold decomposition architecture. They also allow the PBF to vary from one level to the next and in fact even the logic function does not necessarily need to be a PBF as long as its satisfies the stacking property. The architecture presented in Figure 75a represents a set of homogenous GSTFs, restricted to PBFs and therefore is guaranteed to satisfy the stacking requirement. The advantage of filters with the stacking property is they can be implemented using the threshold decomposition architecture and therefore very efficiently in hardware. If the PBF is implemented directly, a variable number of minterms could be implemented using the same hardware. This may be of used to form more complex decision surfaces, but was not explored further in this thesis.

5.8 Chapter Summary

In this chapter image pipelines were found to have greater flexibility in solving problems of practical interest. Pipeline chromosomes do not depend on the size of the training image and the Fitness Metric can be implemented more efficiently than for maximally parallel chromosomes.

Neural Networks can be implemented efficiently as long as small multipliers can be used. Larger multipliers can be demanding on FPGA resources and therefore an alternative known as Morphological Networks was explored. These architectures use a combination of multi-bit addition, subtraction and multiplexer building blocks, which can be implemented efficiently on Virtex FPGAs. The novel contributions of this chapter, relevant to the thesis are:

- Most Neural Network researchers are unfamiliar with the recently introduced Morphological Network. This chapter has made one of the first objective comparisons of these architectures for solving practical AFE problems. Comparisons were made both in terms of the hardware resources required for implementation as well as the quality of AFE algorithm.

- It was found that spectral information alone is insufficient for solving many multi-spectral AFE problems of interest. This is particular true for broad area features such as Urban Areas, where the feature of interest contains many different spectral signatures.

- The flexibility of EA optimization was again demonstrated. The optimization of both neural and morphological networks had similar complexity. This suggests EA can be used to optimize more complex network architectures, or architectural variants as easily as traditional neural networks.

Chapter 6
Evolving Multi-Spectral Networks

Chapter 5 demonstrated how network architectures using multi-bit arithmetic and small multipliers could be efficiently implemented using image pipelines in CC. The flexibility of EA suggests similar components could be used to build other types of networks, which may be better suited to multi-spectral data sets.

This chapter develops exactly that: a novel network node particularly suited to multi-spectral AFE. In terms of design spaces, the node architecture is driven by a combination of:

1. Conventional network architectures and *FPGA efficient* building blocks described in Chapter 5.
2. Software AFE algorithms discussed in detail in the Literature Review.

This chapter presents a detailed description of the node and design motivations. Several developmental experiments were used to help make design decisions. For clarity, these are not reported in this chapter, but interested readers can find examples of these experiments in Appendix A. This chapter does not include assessment and comparison of the network node for AFE. These are reported in Chapter 7, when the node is incorporated in a larger 3-layer network.

Two important roles of AFE in multi-spectral data sets are:

- Modern multi-spectral sensors are now being produced with high spatial resolution. The IKONOS instrument by NASA, which is used in experiments latter in the chapter, has 1m-pixel resolution. To exploit these advances in sensor technology automatic feature identification algorithms must utilize both spectral and spatial information. It is still unclear how these dimensions can be best combined and therefore a good candidate for AFE.

- Sensors produce an order of magnitude more data. A single scene is often imaged at 10 to 20 different wavelengths for multi-spectral sensors [141] and often more than 200 for hyper-spectral sensors. An important part of AFE algorithms is therefore band selection.

6.1 Spatial Processing in Pipelines

The networks implemented in the previous chapter are an excellent example of spectral processing. At each clock cycle, the pipeline combined pixel by pixel, 12 independent spectral channels. Spatial processing requires the pipeline to access multiple pixels from a single channel at the same time. This can be difficult when pixels are being supplied sequentially, however there are several ways it can be implemented:

1) Random access memory: If an image channel is associated with *on-board* memory it is possible to use more complex address generation, and retrieve pixels as they are required. This technique requires careful consideration of the processing pipeline in order to avoid a memory bandwidth bottleneck. Usually pipeline throughput is decreased.
2) Decomposition of Neighborhood processing: For many operations it is possible to decompose the neighborhood processing into a series of row and then column operations. This technique is attractive, however an image must be easily transposed which is often complicated with raster scan images.

A third approach, which is used in this thesis, is to store some of the image locally with on-chip memory. This is illustrated in Figure 76, where image-width length shift registers are used to slide a 3 by 3 neighborhood across the image. Once the row buffers have been filled (an associated latency), a new neighborhood is formed every clock cycle. This means the pipeline efficiency is maintained. This technique is expensive in terms of on-chip memory. However, modern FPGA devices such as the Virtex 2000E used in this chapter have substantial on-chip RAM resources with which row buffering can be implemented.

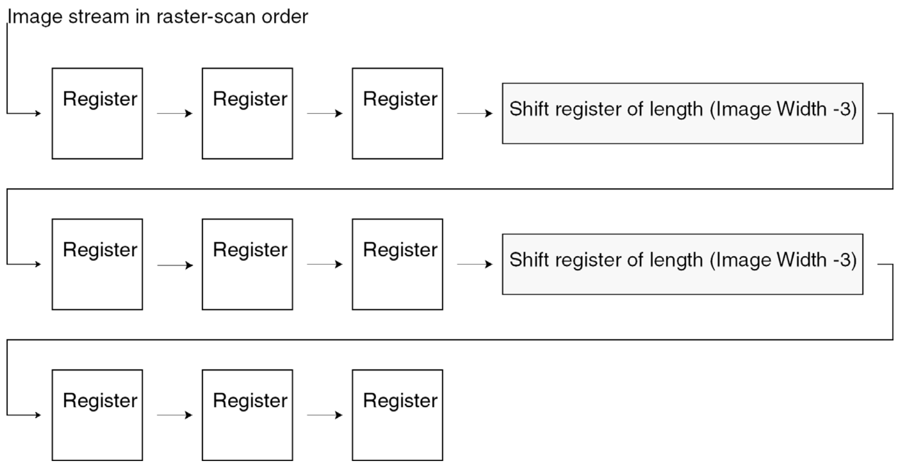


Image stream in raster-scan order
Register
Register
Register
Shift register of length (Image Width -3)
Register
Register
Register
Shift register of length (Image Width -3)
Register
Register
Register

Figure 76: Row Buffering for 3 by 3 Neighborhood Generation

6.2 A Multi-Spectral Network Node

Figure 77 illustrates the multi-spectral network node. The node has four inputs (four multi-spectral bands if applied directly to the data) and one output. The four inputs are first combined using a spectral processor that produces one output. The output of the Spectral Processor is then input to a Spatial Processor.

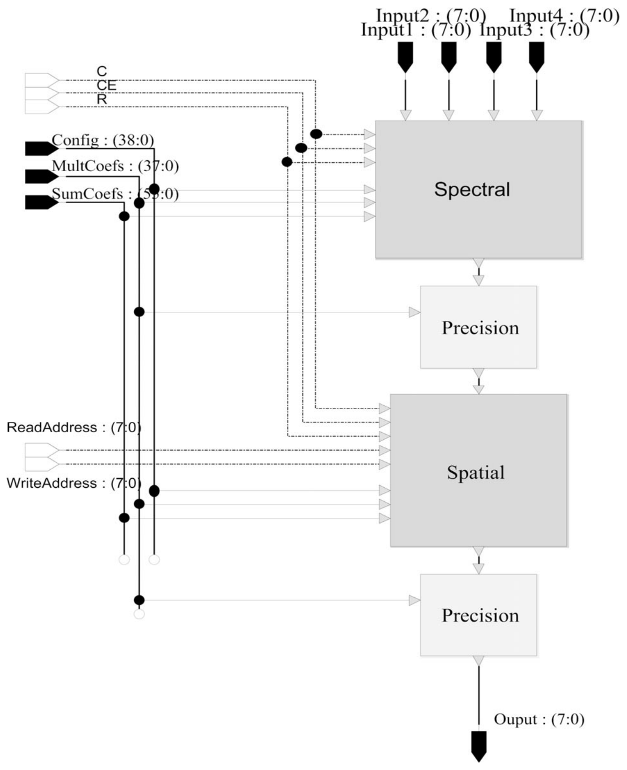


Figure 77: The Multi-Spectral Network Node

The Spectral Processor is described in Section 6.3. The Spatial Processor is described in Section 6.4. The two additional components are used for controlling precision and are discussed in Section 6.5.

The input to the node is assumed to be 8-bit, 2's complement integers in the range –127 to 127. The node is designed to maintain this precision and produce an 8-bit 2's complement output. When the node is used for two-class classification problems, the sign of pixels in the output image dictates what class a pixel is assigned to. Positive pixels are assigned to one class, and negative pixels the other class. Since finding all non-feature pixels is equivalent to finding the feature of interest, two fitness scores need to be calculated: one score for feature pixels being positive and a second score for feature pixels being negative. The host program retrieves both these scores from the CC and chooses the best one.

The node is configured through three on-chip configuration registers, CONFIG, MULT_COEF and SUM_COEF. The configuration registers for the node are described in Table 17. This table indicates the relative number of configuration bits from each of the 4 components: Spectral and Spatial processors and two Precision components. More detailed description of these configuration bits is given as each component is presented.

Register Name	Relative Contribution to 64-bit on-chip Registers				
CONFIG	Spectral 11	Spatial: 30			
MULT_COEF	Spectral 16	Precision 5	Spatial 12	Precision: 5	
SUM_COEF	Spectral 32	Spatial 24			

Table 17: Node Configuration Registers

6.3 The Spectral Processor

The role of the spectral processor is to combine four spectral channels into a single output image. The spectral processor implements a hybrid of the neural and morphological network nodes of the previous chapter. An example of how two inputs are combined in the spectral processor is shown in Figure 78.

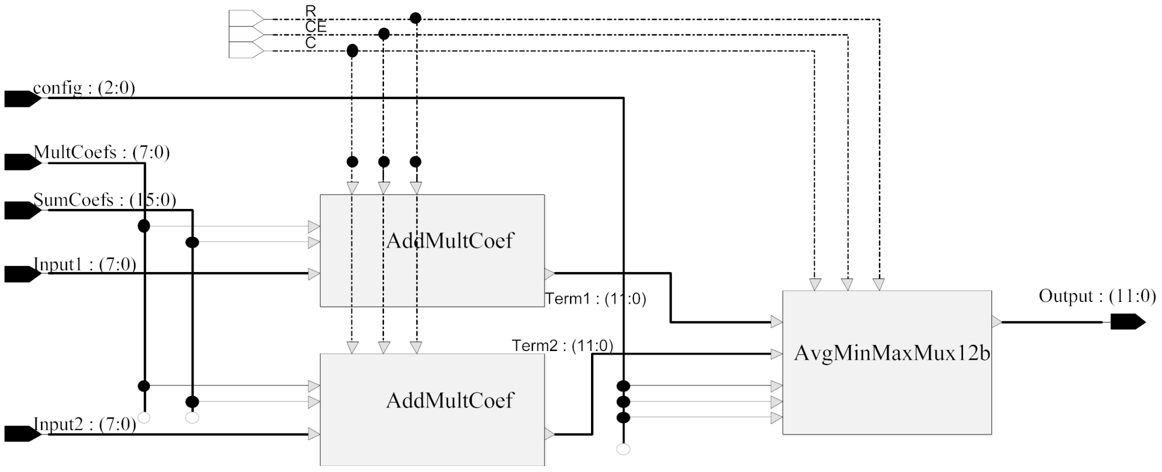


Figure 78: ***2-input Combiner***

Two coefficients are associated with each image plane and are applied in the *Add-Mult-Coef* block in Figure 78. This block implements Equation 16. The sum coefficients have a range between –127 and 127. The Multiplicative coefficients may assume values between –7 and 7.

$$Output = (Input + Sum_{Coef}) * Mult_{Coef} \qquad \textbf{(16)}$$

Once coefficients have been applied, images then pass to the *Arith-Morph-Mux* (AMM) unit. This processing block incorporates the fundamental flexibility of both spectral and spatial processor and is illustrated in Figure 79.

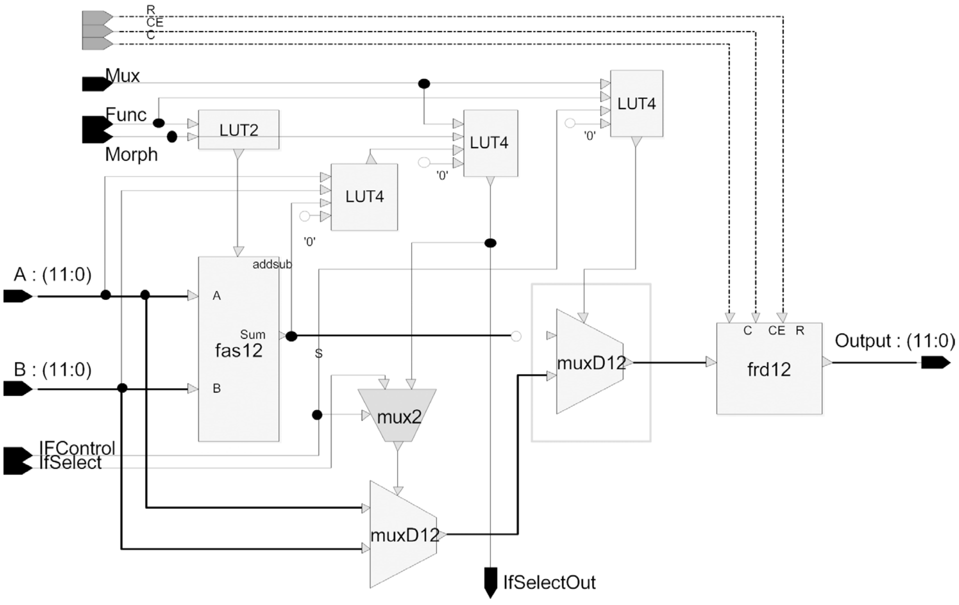


Figure 79: ***Arith-Morph-Mux*** **(AMM) Unit**

The AMM unit is based around a programmable add/subtract unit. This is labeled *fas12* (full adder, subtractor: 12 bits) in Figure 79. With the addition of control logic (implemented in look-up-tables: *LUTs*) and multiplexers (*muxD12*), the AMM unit can be configured to perform several functions of two inputs. A particular function is configured by setting 3 control lines, *Mux, Func and Morph.* These control lines can be seen in the top-left of Figure 79. Table 18 shows the corresponding functions implemented by the AMM unit.

Control Bits			Function
Mux	Func	Morph	Applied to pixels p_1 and p_2
0	0	0	Average $(p_1 + p_2)/2$
0	0	1	Difference $(p_1 - p_2)/2$
0	1	0	Maximum $\vee \{p_1, p_2\}$
0	1	1	Minimum $\wedge \{p_1, p_2\}$
1	*	0	Select p_1
1	*	1	Select p_2

Table 18: Configuring the *Arith-Morph-Mux* Unit

The *frd12* block in Figure 79 is a 12-bit register that introduces pipeline latency. This is a common design practice in pipelines to maintain high clock rates. Additional registers can be used to increase clock rates further.

The final component of Figure 79 is the darkened *mux2*. This component is motivated by the use of the node within a larger network. In this context, the spectral component is used to combine a number of outputs from previous layers. An interesting building block that may be useful to such processing is the IF-THEN-ELSE structure. This structure is often used in GP implementations [55]. Inputs 1 and 2 of the node are evaluated according to an IF expression. Examples of an IF expression include less than, or greater than. THEN and ELSE are implemented through conditional multiplexing. The output is input 3, if the expression is TRUE, and input 4 if the expression is FALSE.

This structure is easily incorporated into the node with the additional darkened multiplexer. This multiplexer has two inputs. The first is the control line, previously used to select the muxD12 multiplexer. The second, *IFSelect* corresponds to a similar control line from a second AMM unit. An additional configuration bit associated with each unit, *IFControl,* is used to choose which of these control lines is used. The complete If-Then-Else structure is implemented by coordinated use of two 2-input Combiners.

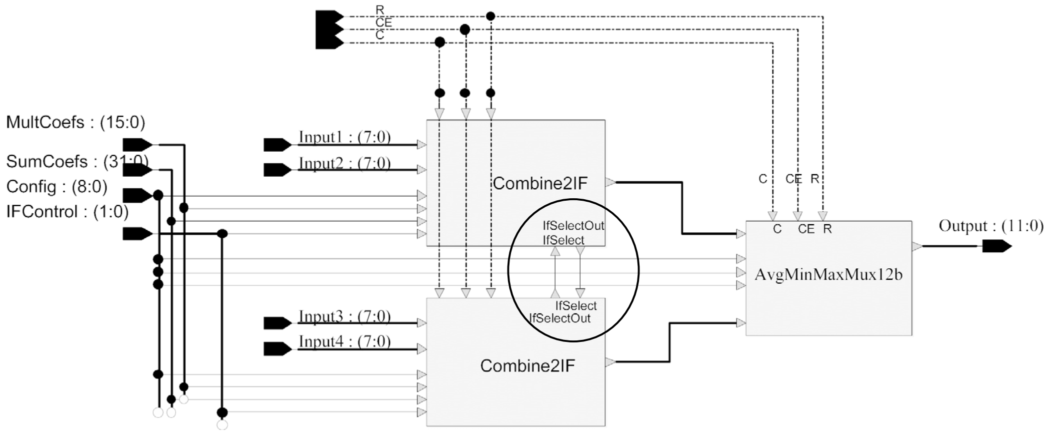


Figure 80: If-then-else Communication between 2-input Combiners

Figure 80 illustrates the complete spectral processor. Two 2-input Combiners of Figure 78 input to a third AMM unit. This is equivalent to a 4 input, 2-layer network. The circle in Figure 80 highlights the additional communication required to complete the IF-THEN-ELSE structure. The original *IFSelectOut* control line in Figure 79 is passed as an output to a second AMM unit. The second AMM unit provides a similar control line back. It is therefore possible to implement two IF-THEN-ELSE structures within the Spectral Processor and hence two *IFControl* bits are present in the Spectral Processor chromosome. The complete chromosome for the 4-input Spectral Processor is shown in Table 19.

CONFIG (0:10)			
1st AMM unit	*2nd AMM unit*	*3rd AMM unit*	
4 bit {Mux, Func, Morph, IFControl}	4 bit {Mux, Func, Morph, IFControl}	3 bit {Mux, Func, Morph}	
MULT_COEF (0:15)			
Input 1	*Input 2*	*Input 3*	*Input 4*
4 bit {Sign, 3 bit INT}	4 bit {Sign, 3 bit INT}	4 bit {Sign, 3 bit INT}	4 bit {Sign, 3 bit INT}
SUM_COEF (0:31)			
Input 1	*Input 2*	*Input 3*	*Input 4*
8 bit {2's comp INT}	8 bit {2's comp INT}	8 bit {2's comp INT}	8 bit {2's comp INT}

Table 19: Configuration Bits for Spectral Components

6.3.1 Application to Feature Space

The spectral processor can be configured to implement both neural network and morphological network functionality that was described in Chapter 5. More importantly, the processor can be configured to be a large number of hybrid-architectures that may be suitable for different problems. The hybrid morphological/neural network nature of the spectral component is most easily seen when it is applied to feature space. Figure 81 illustrates the output from the spectral component, when applied to 3 different problems.

The first problem represents a linear separable feature space, which is easily solved with a linear combination used in neural network architectures. The second problem is an example of the XOR problem [41]. This problem illustrates how the 4 inputs to the spectral processor are combined hierarchically in a 2-layer network. The problem in feature space requires only two inputs, corresponding to the X and Y dimensions of a 2-dimensional feature space. In Figure 81b, the result image shows how two linear combinations are combined to produce the decision boundary. The third problem was designed to illustrate the morphological aspect of the spectral component. In this case, the best decision boundary was found by taking a maximum of two minterms.

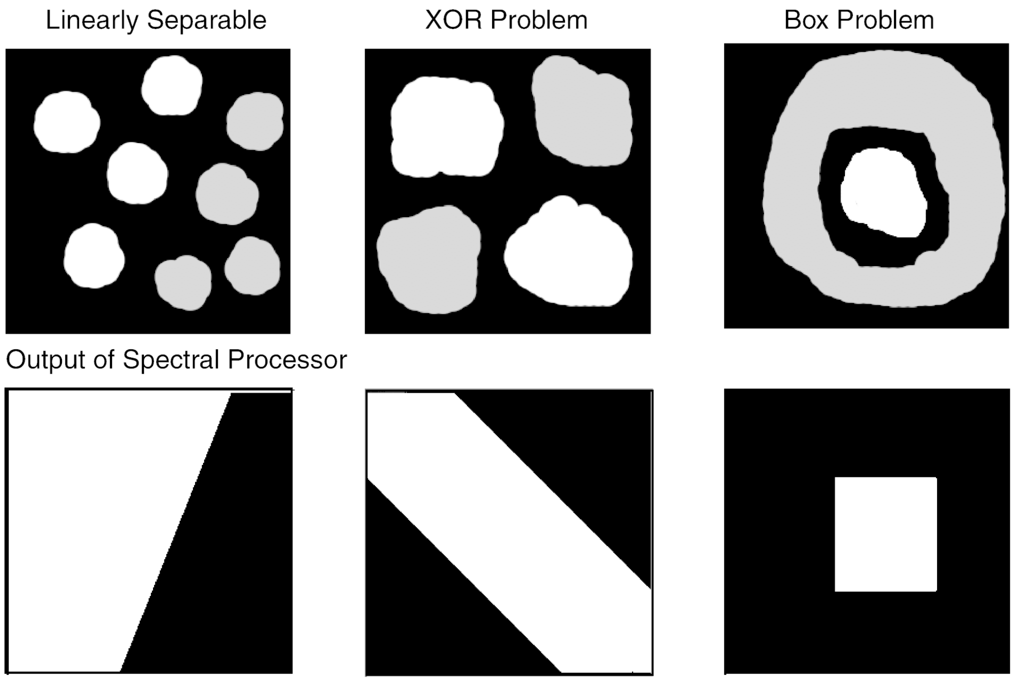


Figure 81: Results on Feature Space a) Linearly Separable, b) XOR and c) Box problems.

6.4 The Spatial Processor

The Spatial Processor is applied to a single input image and implements functions of a 5 by 5 neighborhood. This essentially means that the Spatial Processor takes 25 inputs and produces 1 output. Image Enhancement and Feature Extraction algorithms described in the Literature Review motivate how these 25 inputs are combined. Another major contributor is *hardware reuse*.

Figure 82 depicts the 5 by 5 register array that stores the neighborhood. The input image is supplied to the processor through the row0 input. Four row outputs (on the right of the image) input to row buffers that then supply row1 through row4 inputs. Each register in Figure 82, represents an input to the neighborhood function.

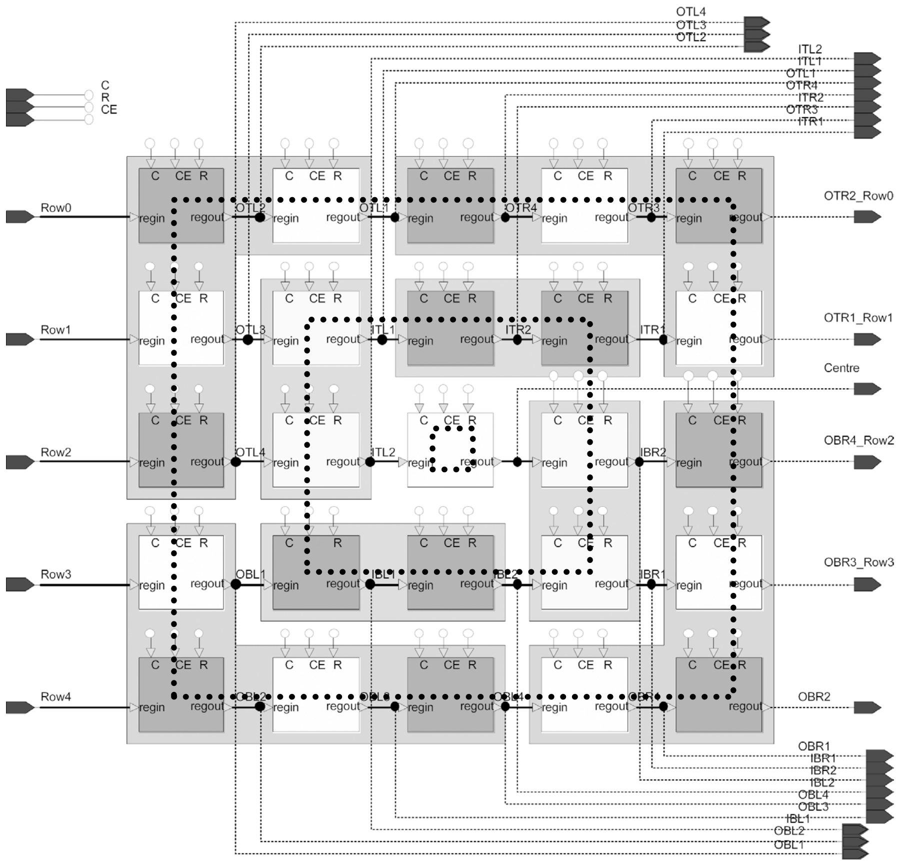


Figure 82: 5 by5 neighborhood of Spatial Operator

The 25 input pixels are combined with a binary network, of *Arith-Morph-Mux* (AMM) units and *Arith-Morph-Abs* (AMA) units. The AMM units were discussed with respect to the Spectral Processor. The AMA unit replaces the multiplexing functionality with an absolute value operation. This is implemented with a conditional complementer, *compl*, in Figure 83. The modified configuration bits are summarized in Table 20.

Control Bits			Function
Mux	Func	Morph	Applied to pixels p_1 and p_2
1	*	0	Absolute Value of Sum $\mid (p_1 + p_2)/2 \mid$
1	*	1	Absolute Value of Difference $\mid (p_1 - p_2)/2 \mid$

Table 20: Modified Configuration of Arith-Abs-Morph Unit

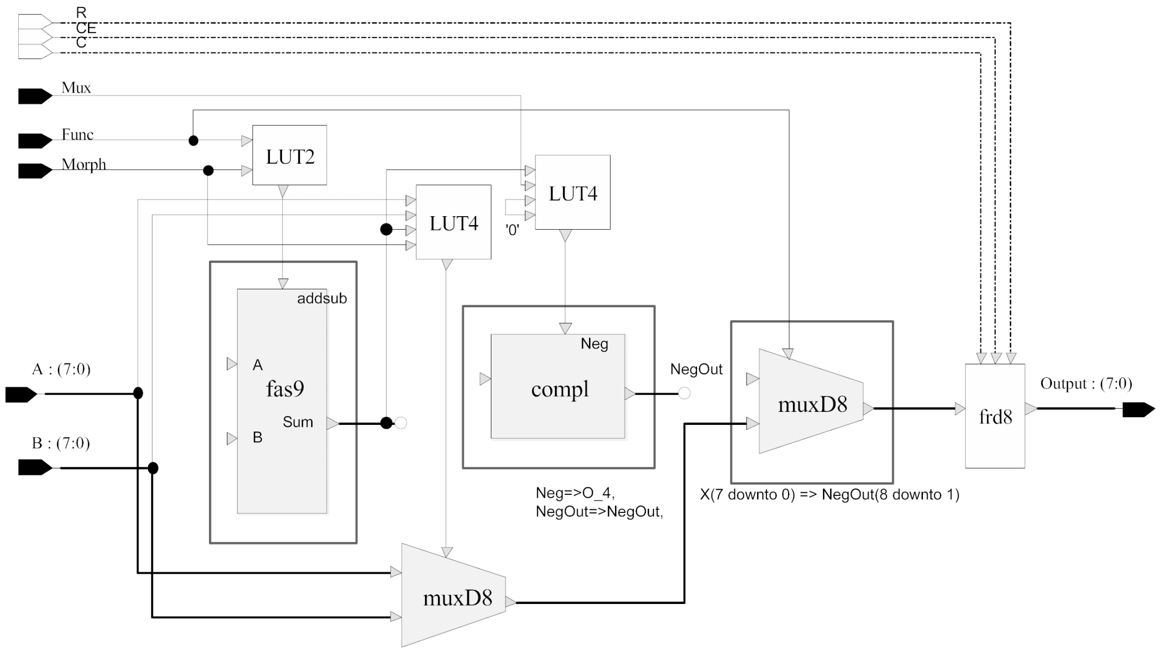


Figure 83: The Arith-Abs-Morph Unit

At the top level, the 25 inputs are first combined into 3 *rings*. These are superimposed on Figure 82. The 5 by 5 ring has 16 inputs and the 3 by 3 ring has 8 inputs. The 3rd ring is simply the center pixel. These 3 values are combined with a spectral processor of 3 inputs. There are therefore both multiplicative and additive coefficients associated with each ring. In this case the IF-THEN-ELSE structure is not included.

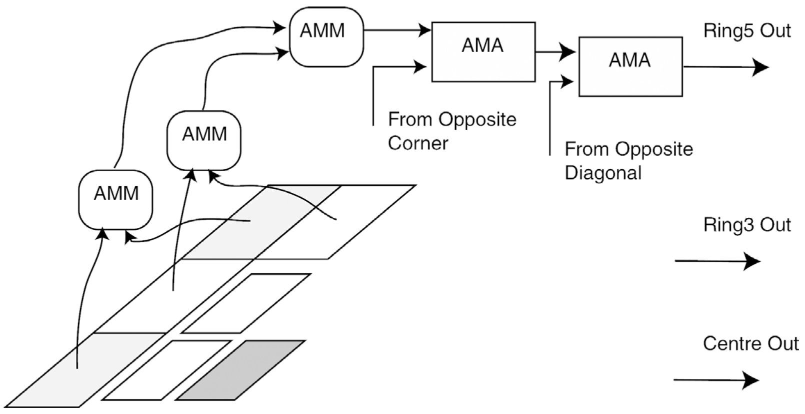


Figure 84: Order of combination for 5 by 5 ring

The order in which pixels are combined in each ring is important. The order for the 5 by 5 case is illustrated in Figure 84. First, pixels in each corner of the ring are combined. In the 5 by 5 case, a 4-input network of AMM units is used. In the 3 by 3 ring, there are only 2 pixels associated with a corner and therefore only 1 AMM unit is required. In both cases, this sub-network can be configured to return the average, maximum or minimum of any subset of pixels in the corner. The corner averages can be used to estimate a gradient by combining opposite corners with the AMA unit. In this case, an absolute value of the difference represents the magnitude of an edge response. Once opposite corners have been combined, the two diagonals that result are combined with another AMA unit. This is most clearly seen in Figure 84.

Each ring can return an average, maximum, minimum or edge response. By associating weights with these rings, a hybrid linear/non-linear spatial filter is implemented. By setting weights appropriately, Gaussian smoothing and simple combinations of Gaussian functions can be implemented. The morphological aspect of the ring combiner can be used to implement, erosion, dilation and morphological range operations. More detailed description of how the Spatial Processor is configured for many Enhancement and Feature Extraction algorithms is found in the next section: *Example Configurations*.

To encourage rotationally invariant operators, and to reduce the size of the search space, only one quarter of the tree is configured. The configuration for the top left quadrant of the tree is used in the other three quadrants. This is a common way of enforcing rotationally invariant structuring elements when optimizing morphological filters [62]. Figure 85 illustrates the technique. Only the top-left portion of the neighborhood with gray background is configured. This configuration is then rotated through the four quadrants. In this example, a particular configuration produces a filter that depends only on pixels that are crossed. In terms of morphology, the structuring element that results can be seen on the right of Figure 85.

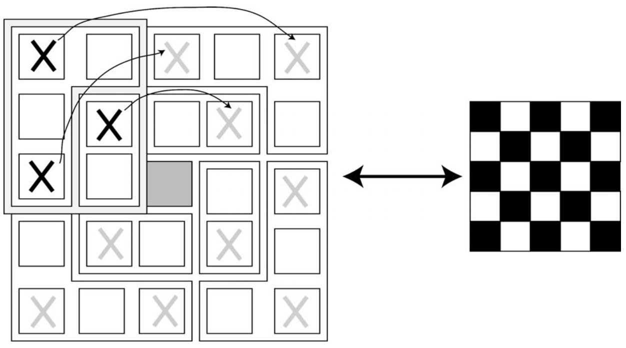


Figure 85: Encouraging Rotational Invariance

The configuration bits for the Spatial Processor are summarized in Table 21. Once the 3 rings are found, the multiplicative and additive coefficients are applied. The 3 by 3 ring is combined with the center pixel using an AMM unit. This is similar to the Spectral Processor. This output is then combined with the 5 by 5 ring to produce the final spatial filter output. Combining the 3 by 3 group with the center pixel before the 5 by 5 group means a flatter spatial convolution can be implemented.

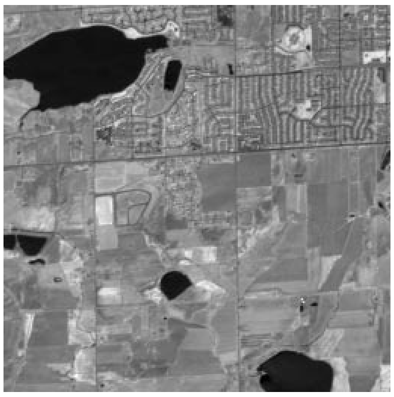
CONFIG (0:29)									
Middle + 5 by 5 Ring	**Center + 3 by 3 Ring**	**3 by 3 Ring**			**5 by 5 Ring**				
		Ring	**Diag**	**Corner**	**Ring**	**Diag**	**Corner**		
AMM	*AMM*	*AMA*	*AMA*	*AMM*	*AMA*	*AMA*	*AMM*	*AMM*	*AMM*
3 bits	*3 bits*	*3 bits*	*3 bits*	*3 bits*	*3bits*	*3bits*	*3bits*	*3bits*	*3bits*
MULT_COEF (0:11)									
Center Pixel			*3 by 3 Ring*			*5 by 5 Ring*			
4 bit {Sign, 3 bit INT}			4 bit {Sign, 3 bit INT}			4 bit {Sign, 3 bit INT}			
SUM_COEF (0:23)									
Center Pixel			*3 by 3 Ring*			*5 by 5 Ring*			
8 bit {2's comp INT}			8 bit {2's comp INT}			8 bit {2's comp INT}			

Table 21: Configuration Bits for Spatial Processor

6.4.1 Example Configurations

This section gives more detailed examples of how the spatial component can be configured to implement Enhancement and Feature Extraction algorithms. These examples explain many of the design decisions made in the Spatial Processor. Figure 86A is the original image to which various configurations of the spatial processor have been applied.

One of the most fundamental image enhancement algorithms, smoothing or low pass filtering, is illustrated in Figure 86B. In this case all AMM and AAM units are configured to find the average of their inputs. The hierarchical summation of inputs towards the center leads to a center bias in the average. This corresponds to a weight of 8 applied to the center pixel, and a weight of 1 for pixels in both 3 by 3 and 5 by 5 rings.



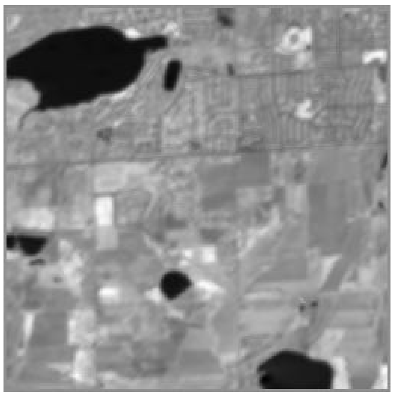


Figure 86: A) Original Image B) and 5 by 5 Smoothed Image

To achieve a semi-flat average the multiplicative coefficients, which combine the 3 rings, can be used. In the case of Figure 86B, weights of 1, 7 and 5 were applied to the center, 3 by 3 and 5 by 5 rings respectively to approximate a Gaussian kernel function described in Section 2.1.1.1 of the Literature Review.

The fundamental Morphological operations, erosion and dilation are also easily implemented with the Spatial Processor. In this case the AMM and AAM units are configured as either maximum or minimum. A key to morphological filters is the shape of the structuring element. In Figure 87, a square-structuring element was used. By configuring some units as multiplexers a variety of structuring elements are possible.

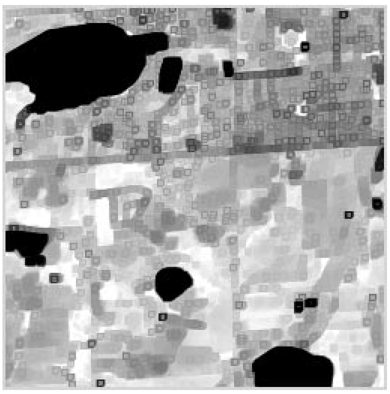
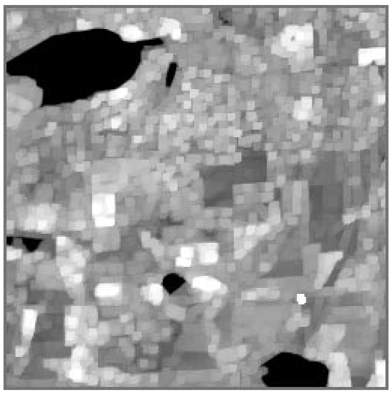



Figure 87: Morphological Erosion and Dilation

By calculating a difference between the averaged corners of the 3 by 3 or 5 by 5 rings, the spatial processor can estimate a gradient along the diagonal. This is similar to the approach used in the Roberts and Sobel gradient operators [6], [7]. To calculate the magnitude of the edge response, the AAM units are configured to calculate the absolute value of the difference. By applying this absolute difference operator in two orthogonal directions (rising and falling diagonals) the total gradient can be estimated with a Maximum [8]. Figure 88 illustrates the result of this approach using the 3 by 3 and 5 by 5 rings.

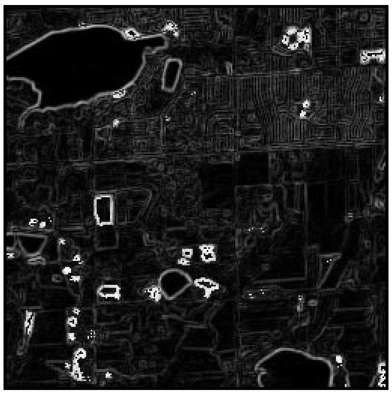
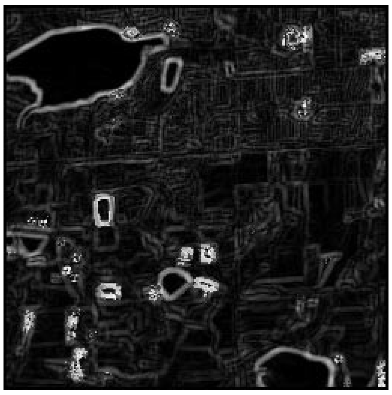



Figure 88: 3 by 3 Edge Detection and 5 by 5 Edge Detection

The Spatial Processor can also be configured to approximate the morphological gradient by calculating the difference between the maximum of the 5 by 5 ring and the minimum of the 3 by 3 ring. This is depicted in Figure 89a. The dual of this operator is the difference between the 3 by 3 ring maximum and the 5 by 5 ring minimum, which is illustrated in Figure 89b.

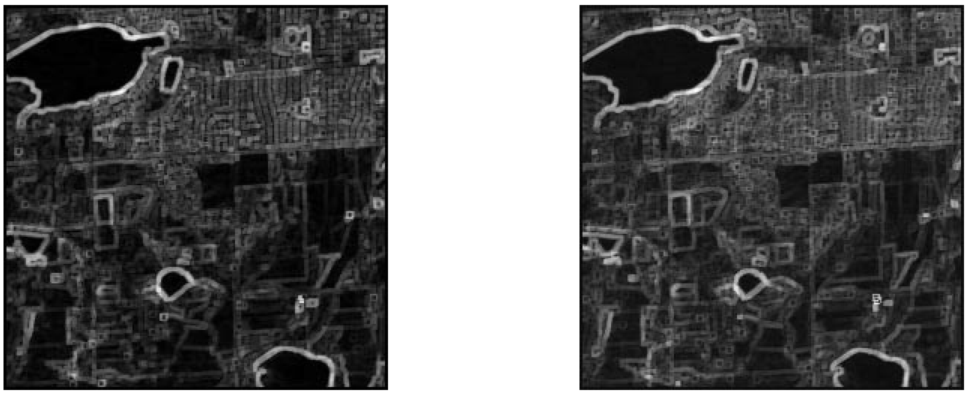


Figure 89: Max-Min Morphological Range and Min-Max Morphological Range

Various texture measures can be implemented with the Spatial Processor. The texture measures of Figure 90 are similar to the rotationally invariant masks by Laws [25]. These texture measures were described in the Literature Review and implement combinations of Gaussian functions (spot detectors) and edge masks [26]. By combining edge responses from one ring with weighted averages of other rings, the spatial processor can effectively implement these types of texture measures. Furthermore, the combination of edge response and smoothing can be optimized by the EA.

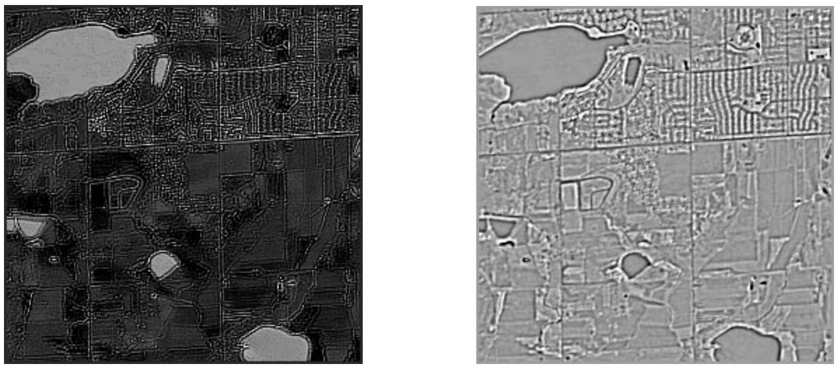


Figure 90: Texture Operators

The combination of linear and non-linear filtering can also be optimized in the spatial processor. For example, generalization of the morphological range leads to linear combinations of ring order statistics. Figure 91A and B illustrate linear combinations of erosions and dilations respectively. This is similar to linear combinations of rank order filters used in L-filters [18] and pseudo-granulomotries described in [38].

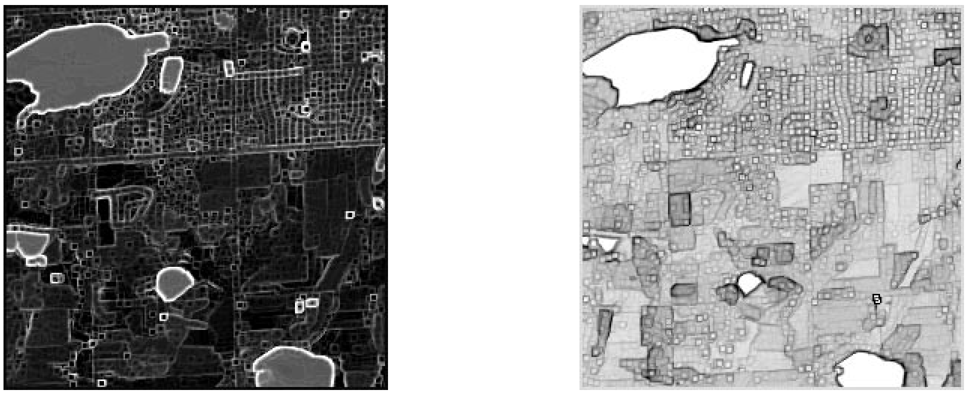


Figure 91: A) Linear Combinations of Erosions B) Linear Combinations of Dilations

6.5 Nodes in Networks

The multi-spectral processing node of the previous section defines a rich search space for AFE in multi-spectral data sets. This network node combines traditional classification components, taken from morphological and neural networks, with well-developed spatial image enhancement and feature extraction algorithms. In this section, additional considerations for using the node in a larger network are described. The reasons why extension to multiple node architectures is desirable:

1. In multi-spectral imagery, it is possible that more than 4 input channels would be required to solve a problem accurately. The width of the network is defined by a number of nodes in parallel. Each node has 4 independent inputs and therefore more multi-spectral channels can be processed.

2. It is likely a single node cannot solve a general set of more complex problems. The depth of the network is defined by a number of nodes in series. By applying a number of nodes in succession more complicated algorithms can be implemented.

Section 6.5.1 describes the Precision component. This was illustrated in Figure 77 at the start of this chapter, but is yet to be described. Section 6.5.2 then describes how a channel chooser can be implemented, which allows the network to be applied to variable size data cubes.

6.5.1 The Configurable Precision Unit

It is desirable to maintain a consistent bit-width between node input and output so that nodes can be easily cascaded to form networks. Input to the node is assumed to be 8-bit 2's complement numbers. The multiplicative weights, used to combine inputs in the Spectral Processor, and the center, 3 by 3 and 5 by 5 rings in the Spatial Processor produce 12-bit outputs at full precision.

One solution is to constrain the output to either the least significant or most significant 8-bits and use the EA to find solutions that can perform well under this constraint. Since multiplicative weights can be positive or negative, it is possible that coefficients could be found that produce a final result within the desired dynamic range. An alternative is to provide a means to dynamically select 8-bits from the 12-bit output from one chromosome to the next. This is the approach used in this thesis.

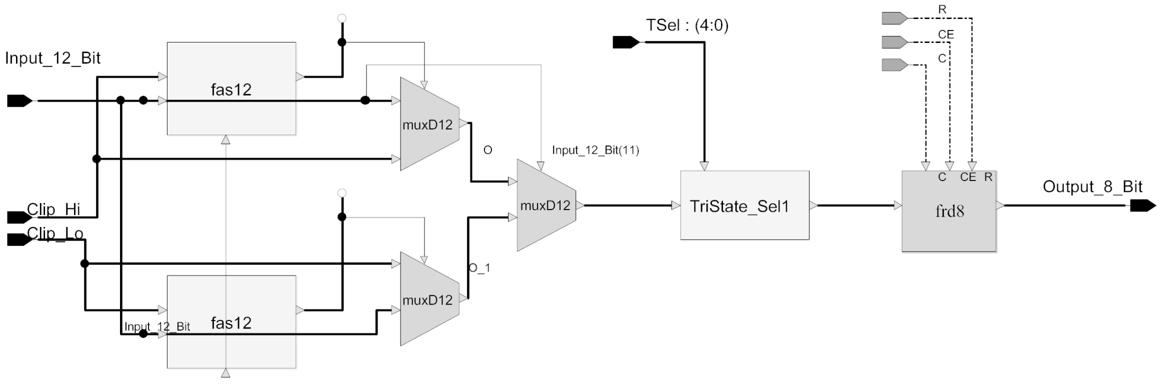


Figure 92: The Configurable Precision Unit

Figure 92 illustrates how this is implemented within the node. An 8-bit wide tri-state bus (the *TriState_Sel* unit) is used to multiplex the twelve bit input data. By setting control lines, this unit effectively divides by 1,2,4,8, or 16. The 12-bit add/subtract units (*fas12*) and 12-bit multiplexers (*muxD12*) depicted on the left of Figure 92 are used to clip the 12-bit input at the appropriate values. These values are loaded by the host to on-chip configuration registers: *Clip_Hi* and *Clip_Lo*. The values of these registers are determined by the value of the *TSel* tri-state control lines, which are included in the node chromosome. The relationship between *TSel* and the *Clip_Hi, Clip_Lo* registers is shown in Table 22. Note, the values of the Clip registers could be generated on-chip since they are determined by *TSel*. They are downloaded by the host program to simplify the implementation. It is also possible that they could be used in the future to implement a tunable banded threshold.

Divider	TSel (Tri-state Select)	Clip_Hi Register	Clip_Lo Register
0	1 (11110)	127	-127
1	2 (11101)	255	-255
2	3 (11011)	511	-511
3	4 (10111)	1023	-1023
4	5 (01111)	2047	-2047

Table 22: Configuration of the Precision Unit

There is an additional component in the Precision Unit that cannot be seen in Figure 92. It is a programmable absolute value operation that is controlled by a single bit included in the node chromosome. The absolute value is necessary for some of the spatial processor configurations such as zero-sum linear convolution texture measures. In this case, the absolute value of the output image contains the most useful information. This is similar to using the absolute value to estimate the magnitude of the edge response discussed earlier.

6.5.2 Incorporating Band Selection

One of the main motivations in applying AFE to multi-spectral data processing is to select the appropriate spectral channels of the image cube that are relevant to a particular problem. In our case, it is desirable that a node input has the potential of receiving input from any image channel. To accommodate a variable number of channels, the tri-state bus resources of the Virtex FPGA are used to implement large multiplexers. Local memory resources of the Firebird RCC dictated an upper limit of 12 8-bit channels. The 12-input channel chooser is illustrated in Figure 93.

There are 8 tri-state buses associated with each node input: 1 tri-state bus is associated with each bit in the 8-bit data. Each bus may be driven by any 1 of the 12 input channels which is determined by setting the tri-state control lines, *TSel*, in Figure 93. *TSel* is mapped to an on-chip register and is included in the network chromosome.

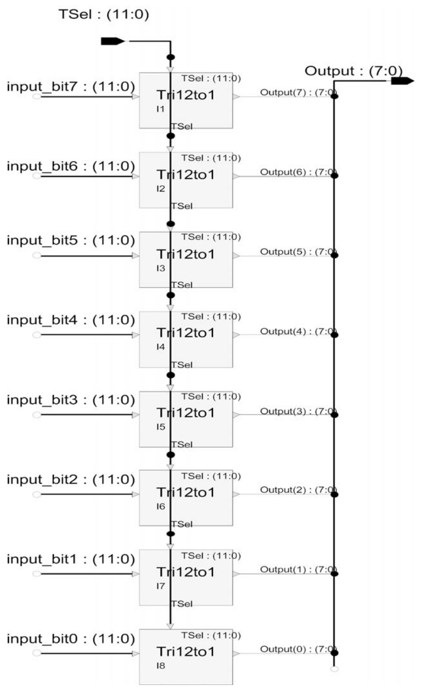


Figure 93: The 12-band Channel Chooser

6.6 Evolutionary Algorithm

The Multi-Spectral Network Node, and its configuration registers have been described in detail. How the EA is applied to these configuration registers is now described.

6.6.1 Representation

Some components of the configuration are naturally suited to binary string representation. Others, such as the additive and multiplicative coefficients are best evolved on an arithmetic level. Table 15 describes the chromosome components, how they relate to the configuration registers, and how genetic mutation is applied.

Both Spectral and Spatial Processors have BITSTRING and INT components. The additive coefficients are stored in the hardware registers using two's complement representation. For the multiplicative coefficients, a sign bit is used (MSB). The Precision component is represented by an UNSIGNED integer, which is translated into tri-state control lines and clip-registers described previously in Table 18. The Band Selection chromosome is also an UNSIGNED integer in the range {1:NumBands}, with a maximum range of {1:12}.

Chromosome in Software	**Configuration Registers in CC**	**Mutation Strategy**
Spectral and Spatial Processors		
BIT Func BIT Morph BIT Mux/Abs	Binary bit Binary bit Binary bit	Bit flip Bit flip Bit flip
INT Sum_Coef[4]	Two's Complements form of: Sign(Sum_Coef[i])*$2^{\lvert Sum\_Coef[i]\rvert}$	±1 in range {–7 to 7}
INT Mult_Coef[4]	Sign Bit Representation of Mult_Coef[i]	±1 in range {–7 to 7}
Precision Unit: 2 Per Node		
UNSIGNED Divider	Tri-state control lines and Clip Registers (Table 18)	±1 in range {0 to 4}
BIT Abs	Binary Bit	Bit flip
Channel Chooser: 4 Per Node (Input layer nodes only)		
UNSIGNED Band	Tri-state control lines	±1 in range {1 to Number of Bands in Training Data}

Table 23: Software Chromosome for Spatial-Spectral Processing Node

Similar to the hardware implementation, the software chromosome is stored hierarchically in a number of objects. This is illustrated in Figure 94. The Mutation and Crossover commands originate in the *Population* object in the host program presented in Chapter 5. They are passed to a *Network* object, which then distributes them to particular *Node* objects, which themselves are decomposed into their constituent parts.

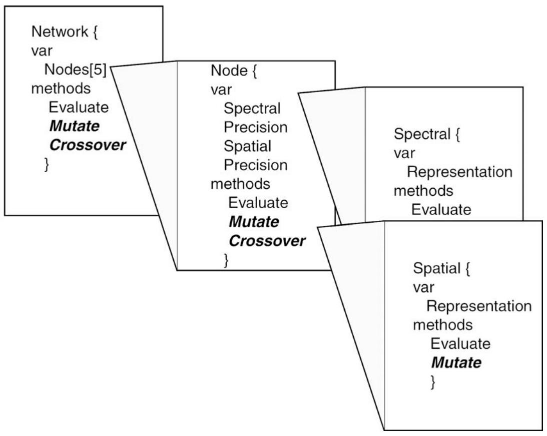


Figure 94: Decomposition of Representation in Chromosome Objects

6.6.2 Mutation

When nodes are incorporated within networks, each node has an equal chance of being mutated. Once a node has been selected, mutation can be applied in a variety of ways most easily visualized as a mutation tree. For each Mutate command sent by the *Network* object there is a probability of a particular branch being taken. This is illustrated in Figure 95. Within each component, mutation points are chosen with equal probability. The choice of probabilities is fairly arbitrary and is based on familiarity with the chromosome and experimentation.

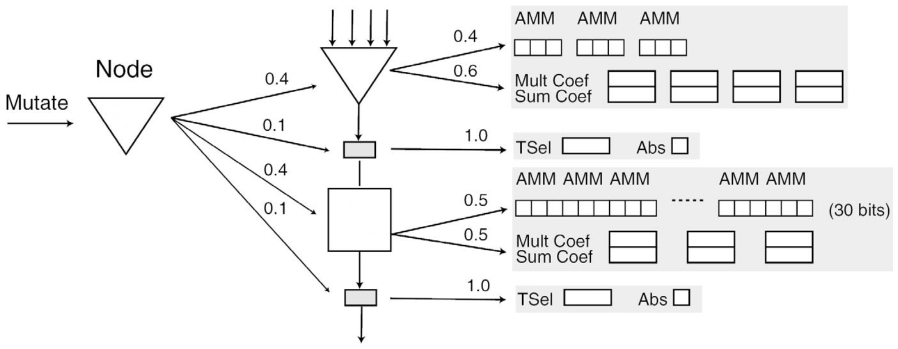


Figure 95: Hierarchical Implementation of Mutation

For nodes on the input-layer of a network, the configuration for 4 Channel Choosers is also included within the node chromosome. If an input-layer node receives a Mutate command, there is a 20% chance that this will randomly mutate one of the 4 inputs to the node. The remaining 80% of the time, mutation is applied according to Figure 95.

6.6.3 Crossover

Crossover within the node is applied in a similar way to mutation and is illustrated in Figure 96. In this case there is a 50% chance that the crossover is applied to the spectral component and 50% chance the spatial component. Within these components, crossover points are chosen with equal probability.

The crossover operator for a two-layer network adds another level to the hierarchy and is illustrated in Figure 97. First, one of 3 crossover points in the first layer is selected. Then there is a 70 % chance that all 1st layer nodes to the left of the crossover point are swapped between parents. The remaining 30% of the time, crossover occurs with all nodes to the left, but then crossover is also passed into the node. The crossover point within the node is then chosen according to Figure 96.

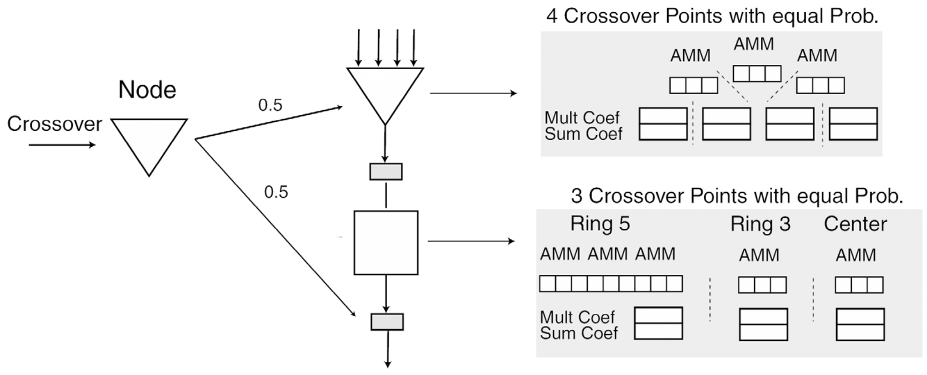


Figure 96: Hierarchical Implementation of Crossover

Regardless of where the crossover occurs in the 1st layer, there is a 50% chance that the 2nd layer node is swapped between the parents. Again of this 50%, half the time the entire node configuration is swapped and the other half of the time, the crossover point is selected within the node.

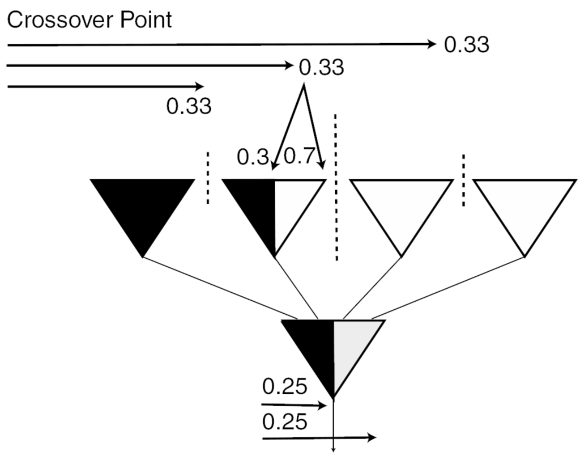


Figure 97: Crossover in the Network

6.7 Experiments with Two-Layer Networks

Hardware design, like software, is an incremental process. The first network to be implemented was a two-layer / 5 node network illustrated in Figure 98. This network was implemented to make preliminary investigation of the network approach and verify implementation accuracy before synthesis, and place-and-route times became too large. This section describes an experiment that dictated several design choices in the larger 3-layer network described in the next chapter.

The top-level architecture described in Chapter 5 was used for the implementation and Channel Choosers were not implemented. The training data is limited to 4 image channels, which are tied directly to the 4 inputs of the node. For the two-layer network, each node received the same 4 inputs. The two-layer network was able to meet 66Mhz-timing requirements. A more detailed discussion of implementation is given in Chapter 7 with respect to the larger 3-layer network that was eventually implemented.

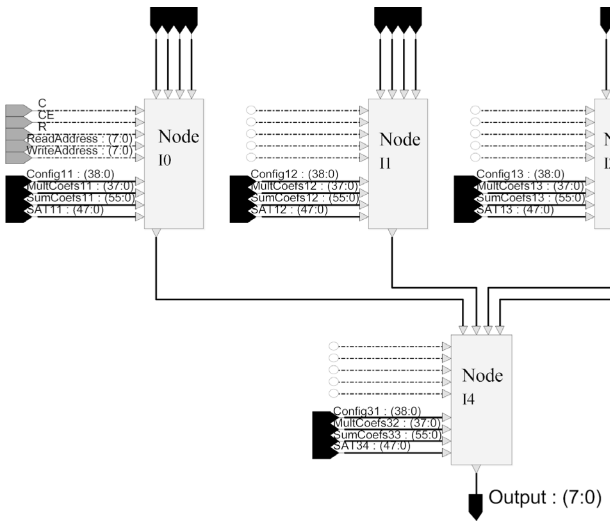


Figure 98: The Two Layer / 5 node network

6.7.1 Experiment

The chromosome for the 2-layer network is approximately 5 times as big as the node chromosome. It was therefore of interest to compare EA optimization of the 2-layer network with 5 independent optimizations of the single node. In the second case, four evolutionary runs are performed on the training data, resulting in 4 node configurations and 4 output images. A 5th node is then optimized using the 4 output images as training data.

The training image for the comparison is shown in Figure 99. It was taken by the IKONOS instrument and has 4 channels (3 visible and near infra-red). The IKONOS instrument has 1m spatial resolution and therefore spatial information is essential in obtaining accurate AFE. The problem of Figure 99 is to find the roads. This is thought to be a relatively easy problem for which networks should be able to obtain relatively high scores. Since fitness is expected to improve substantially in the evolutionary run, this problem is a good candidate to test EA strategies. The IKONOS data sets were used extensively in developmental experiments for the node, examples of which can be found in Appendix A.

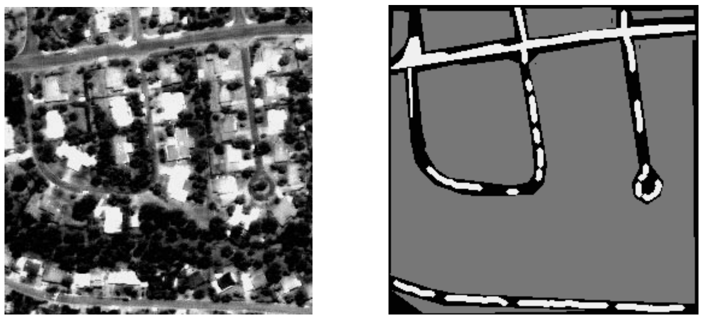


Figure 99: IKONOS Road Finder Training Data and Target Classifications

6.7.2 Evolutionary Algorithm

The chromosome used in the independent evolution is the node chromosome previously described. For the 2-layer network, the chromosome is made up of 5 node chromosomes and is therefore 5-times larger. How mutation is applied to both node and network was also described. For both network and single node experiments, a generational EA with elitism is used with the schedule summarized in Table 24.

EA Parameter	**Number**
Population Size	300
Number of Generations	50
Parents	200
Reproduction	
Elite	20
Mutated Elite	60
Crossover	100
Mutation	100
Random Generation	20

Table 24: EA Parameters Used

6.7.3 Results and Discussion

Figure 100 shows the output when nodes were independently optimized. The best first layer score was 961 and this was increased to 970 by the output node. When the network was optimized as a whole, the final output reached a score of 961, the same score achieved by a single node using independent evolution. This is illustrated on the left of Figure 101 Note, the training time for the single node experiment is 5 times as long as the training time for the network. This is because the single node experiment evolved 5 nodes in series. For this reason, the network was also evolved for 5 times the number of generations. The result of this optimization is seen on the right in Figure 101, and reached a score of 964.

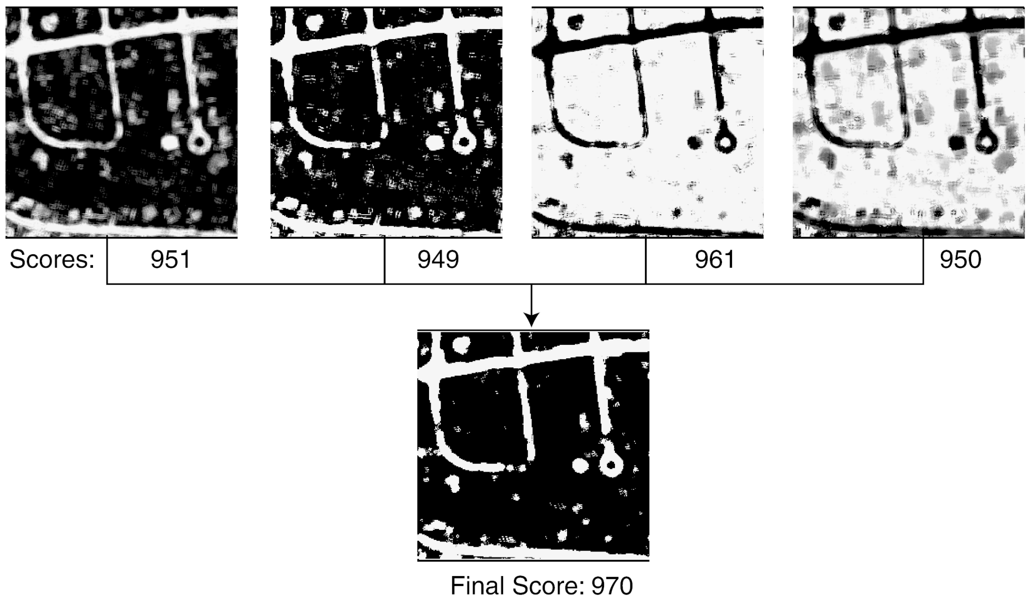


Figure 100: Results of Independent Node Evolution

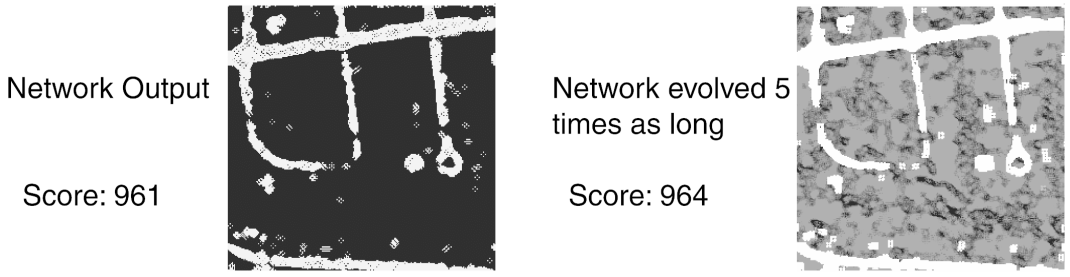


Figure 101: Result of Network Evolution

It can be seen that the independent evolution of multiple nodes outperformed the network evolution. It is likely this result is due to the much larger search space associated with the network. Additionally, many components of the network chromosome are dependent. A first layer node may perform extremely well but will not receive reward if the output node is not properly configured. This implies a more complex evolutionary search technique is required. This problem has appeared often in literature and many researchers have proposed more complex evolutionary strategies. Some of these are described in the next Chapter.

An immediate solution, suggested by the two-layer network experiments, is to calculate a fitness score for each node independently. This would enable the independent evolution of 4 network nodes to be performed in parallel by using the first layer of the two-layer network. This motivated the implementation of multiple fitness units in the three-layer network: one for each node.

In addition, it can be seen that using independent evolution allowed the output from each node to be retrieved and inspected. This is potentially a very useful thing; especially for understanding how an optimized network solves a particular AFE problem. This dictated the use of additional local memories on the Firebird CC for storing the output from each node and is described in more detail in Chapter 7.

6.8 Chapter Summary

This chapter has presented a novel network node suitable for multi-spectral image processing. The node extends the architectures of Chapter 5, and defines a search space for AFE which is both:

1. Hardware efficient in terms of FPGA implementation: The node is built from small arithmetic units and multiplexers that are well matched to the Virtex FPGA.
2. Well motivated for the problem: The node architecture draws considerably from human inspired design spaces that are used to solve practical image-processing problems in literature.

Properties of the node that are novel include:

- A combination of spectral classification techniques with spatial enhancement and feature extraction algorithms, in a self contained, modular design. This means *hybrid* feature extraction/classification architectures that are scalable, inherently parallel and easily implemented.
- A novel combination of neural network, morphological network and IF-THEN-ELSE in a hybrid *Morphological-Linear* building block for spectral classification.
- A novel use of the hybrid *Morphological-Linear* building blocks for spatial filters. The family of linear spatial filters seen in convolutional neural networks can be implemented. Additionally, the equally useful family of non-linear morphological spatial filters can be implemented.
- Also, the unique organization of the building blocks in the spatial processor encourages rotationally invariance and allows extraction of edge and texture information.

Chapter 7

Experiments with POOKA

In this chapter the multi-spectral node developed in Chapter 6 is used in a 3-layer network implementation known as POOKA[5]. This chapter evaluates the network both in terms of efficiency of implementation, as well as quality of AFE algorithm.

The preliminary experiments reported at the end of Chapter 6 dictated several extensions to the top-level architecture and host program. The extensions to the Fitness Evaluator architecture are reported in Section 7.1. The revised host program and Evolutionary Algorithm are then presented in Section 7.2. A detailed evaluation of the FPGA resources and measurement of speed-up compared to software is made in Section 7.3.

The next two sections are experimental. The first experiment, in Section 7.4, applies POOKA to the 3 multi-spectral test problems of Chapter 5. A comparison is made of EA strategies for optimizing the network. In Section 7.5.1, POOKA is compared to a number of other AFE techniques using a previously published test-set. The other techniques include the advanced spatio-spectral GENIE software system described in the Literature Review as well as more conventional spectral classifiers: The Spectral Angle Mapper and a Maximum Likelihood classifier. Limitations of the Section 7.5.1 test-sets lead to more detailed exploration of POOKA's generalization performance and convergence behavior in Section 7.5.2.

[5] Concurrent to this thesis several software AFE systems were developed at Los Alamos National Laboratory. The various systems adopted names from mythology including GENIE, AFREET and MARID. POOKA is the name of a mythical ghost horse, and with the promise that hardware could provide significant horse-power to AFE problems, the name has stuck!

This chapter concludes in Section 7.6 with a novel application that is made possible by the revised top-level architecture. This application involves using POOKA to evolve algorithms for multiple features of interest in parallel.

7.1 Top-Level Architecture

POOKA is a 3-layer, 9-node network implementation that is illustrated in Figure 102. Similar to the two-layer network, there are 16 independent inputs to the network and therefore 16 Channel Choosers are required. Each node in the second layer receives input from the four outputs of the first layer. It can be seen in Figure 102 that the order of inputs for second layer nodes is kept constant. This means the first input of the I5 node is the same first input for I6, I7 and I8 nodes.

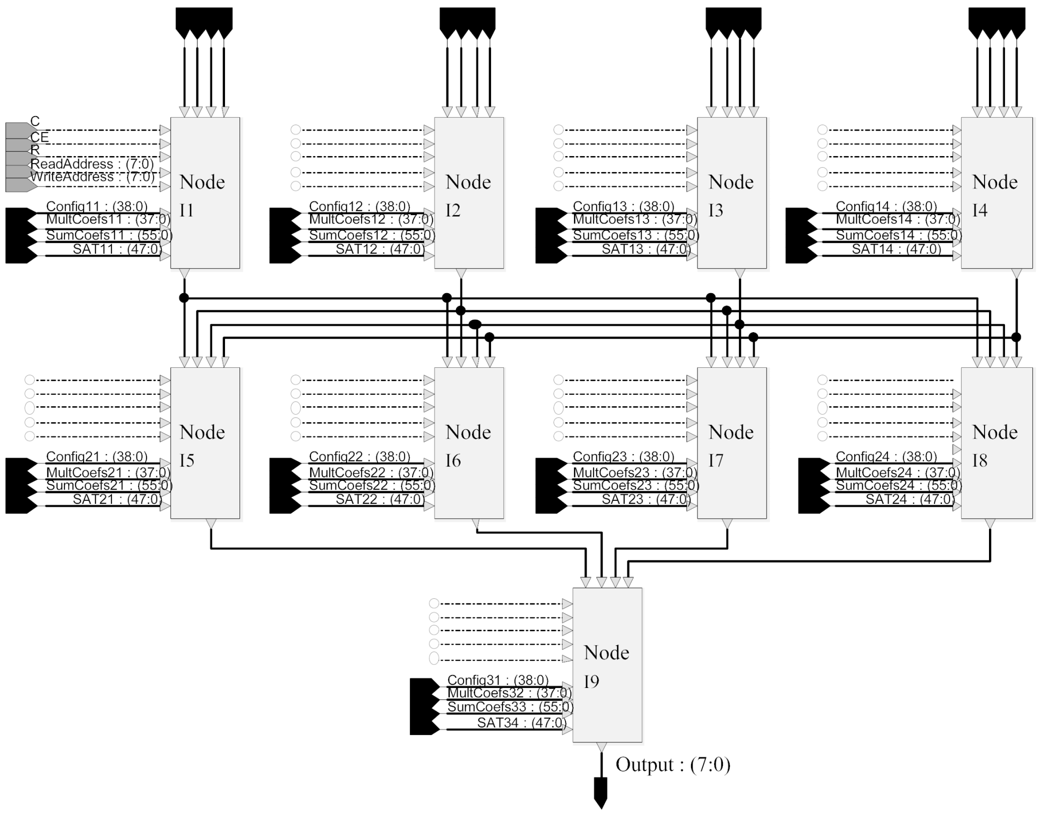


Figure 102: The 3-Layer Network

Not shown in Figure 102, is the fact that all node outputs are made available to the top-Level architecture. This can be seen in the revised Top-Level architecture in Figure 103. The output from each node is passed through to the top-level for two reasons:

1. The node output is sent to on-board memory. The host program can then retrieve the individual node outputs. In the two-layer experiment, this was

found to be useful in understanding how the network is solving a particular problem. In Figure 103, it can be seen that the outputs from the first 2 layers of the network are sent to another Firebird local memory, *Local Memory 4*.

2. Each node output is also sent to an independent *Fitness Metric* unit. This was also motivated by the experiment using two-layer networks. In Figure 103, the additional Fitness Metric units are implemented within the *Fitness8* component.

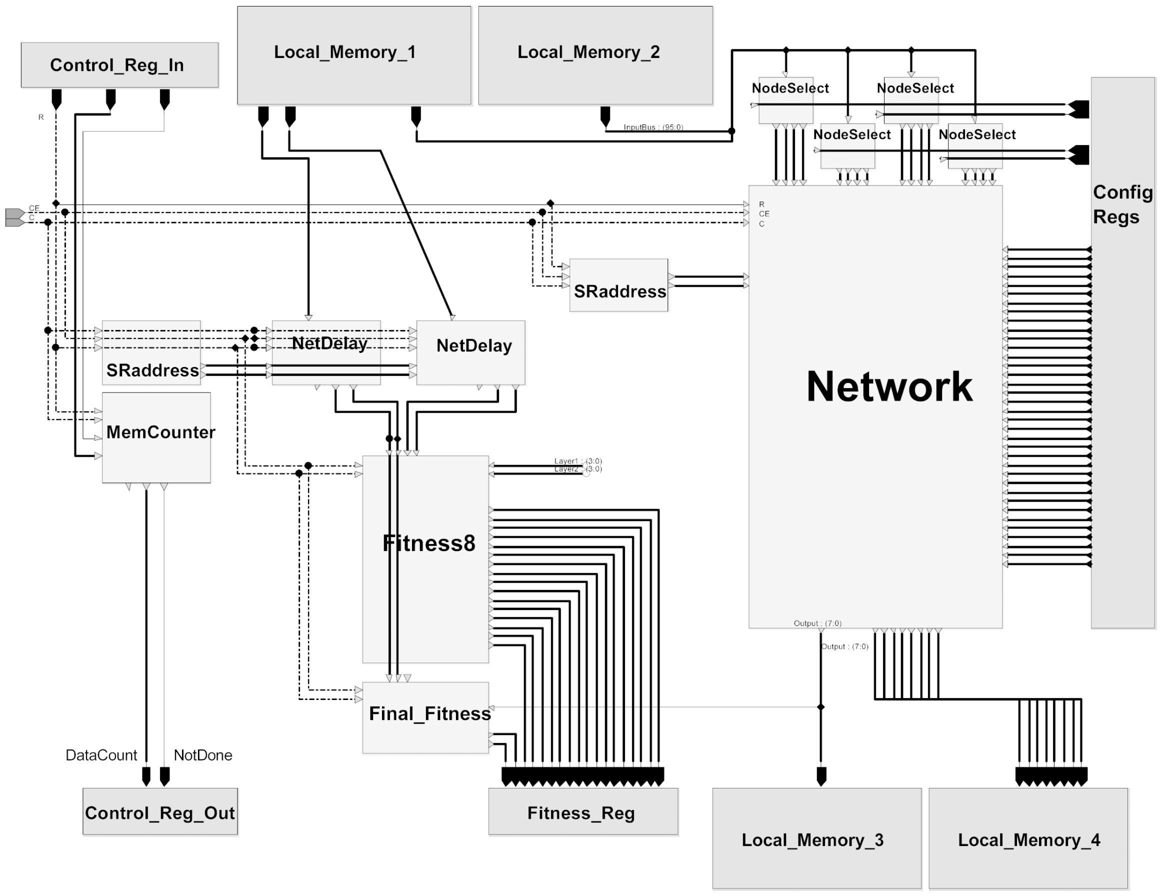


Figure 103: Top-Level Architecture for 3-layer Network

Figure 103 also illustrates the addition of 4 *Node Select* components, one for each node in the first layer of the network. There are 4 Channel Choosers implemented within each *NodeSelect,* one for each input to the node. A maximum of 12 spectral channels, stored in *Local Memories 1 and 2,* input to the Channel Choosers.

Another extension, which is used in experiments of Section 7.6, is the use of multiple target classifications (truth and weight images). A total of 9 target classifications are loaded to *Local Memory 1*, one for each node of the network. Each target classification is a 2-bit pixel image, 1 bit indicating truth and 1 bit indicating weight. Therefore, a data path of 18-bits is required to *Local Memory 1* to implement this extension. The 9 target classifications are supplied independently to the 9 *Fitness Metric* units. Usually the target classification associated with each node is the same. In this case, 9 copies of the target classification are stored in *Local Memory 1*.

7.2 Host Program and Evolutionary Algorithm

Several extensions to the host program and evolutionary algorithm were also implemented. There is a large quantity of work, directed at evolving neural network architectures, partly due to the rapid increase in search space size as the network grows **[76]**. For the POOKA architecture, some of these techniques are more appropriate than others. An immediate observation is made: calculating an independent fitness for each node suggests implementing multiple populations. The revised host program architecture is illustrated in Figure 104

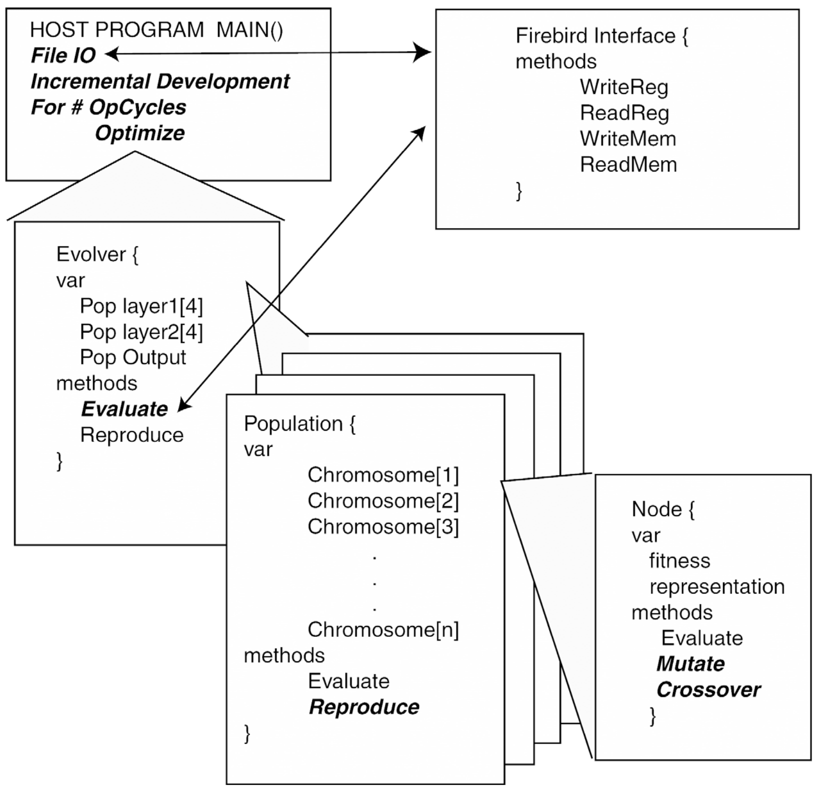


Figure 104: Host Program using multiple populations

At the top-level, the host program is very similar to that described in Chapter 5. A new object *Evolver* is introduced, which instantiates 9 independent *Population* objects, one for each node in the network. For each evaluation the *Evolver* chooses a particular node from each population. These nodes are configured in the network *Generalized Chromosome* and evaluation takes place as normal. The *Evolver* then retrieves the fitness score associated with each node, and assigns it to the appropriate chromosome in the *Population* objects. Each *Population* maintains a population of *Node* chromosomes independently. Reproduction and genetic operators are only applied to nodes within the same population. The Mutation and Crossover node operators described in Chapter 6 are used. By keeping populations of nodes independent, specialization of nodes is encouraged.

Multiple populations and independent Fitness Metric units allows efficient implementation of *Incremental Learning* techniques that were described in the Literature Review. Incremental evolution of the POOKA network in Figure 102 is a 3-stage process:

1. The four 1st layer nodes are evolved in parallel in four different populations. Since a fitness metric is calculated on the output from each node, these populations can be evolved independently.
2. In the second stage, the best 1st layer nodes in each population are configured and remain fixed. The 4 nodes in the 2nd layer are then evolved independtly in 4 populations.
3. In the third stage, the best 2nd layer nodes are also configured. Both 1st and 2nd layer nodes remain fixed and only the output node is evolved.

To maximally utilize the fitness evaluator resources, all 9 nodes should be involved in evolution at all times. This is not possible with the *Incremental Learning* approach, and some nodes remain fixed while others are evolved. It is possible to evolve higher layer nodes while lower-level nodes are evolved. This means the 1st, 2nd and 3rd layers are evolved in Stage 1. Only the 2nd and 3rd are evolved in Stage 2 and just the 3rd layer in Stage 3. This is illustrated in Figure 105 for clarity. The arrows in this figure indicate that optimization of the node configuration is based on the nodes output.

This can produce unpredictable fluctuations in the higher-layer scores, since the data they are supplied with can vary from one evaluation to the next. At the start of the network evolution there is another affect. That is, reward is given to nodes that simply pass the data on. They are rewarded for the high scores from the lower layers and therefore not the processing they perform.

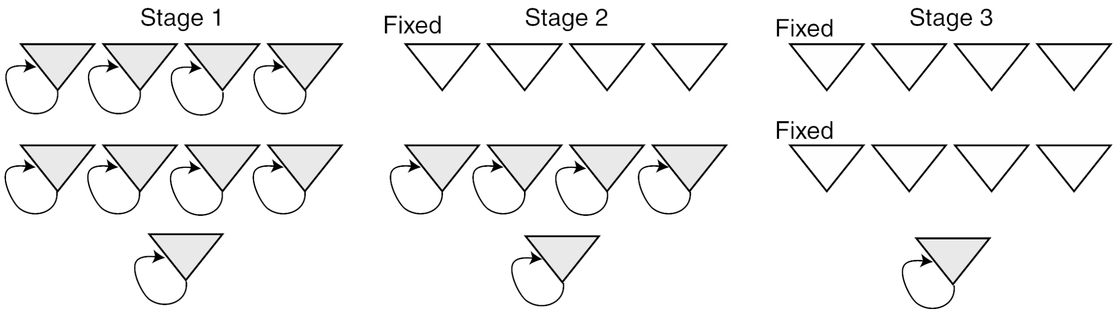


Figure 105: 3 Stage Incremental EA

After *Incremental Evolution*, a variable number of *Optimization Cycles* are applied. This is motivated by the fact that nodes that may not score well individually can be very useful within the network and in fact may lead to better scores in the final output. This is the main reason why competition is not the only factor in network optimization, and co-operative behavior is desired. Various mechanisms have been suggested for implementing this with varying levels of complexity. The Optimization Cycle approach used in POOKA is most similar to the method suggested in [73].

In the Optimization Cycle, nodes receive reward based on the final network output. There are 9 stages to the optimization cycle, 1 for each node in the network. The network is configured with the best nodes from each population that were found in the incremental development phase. Each node in the network is then evolved in turn using the fitness calculated on the final network output. This is illustrated for clarity in Figure 106.

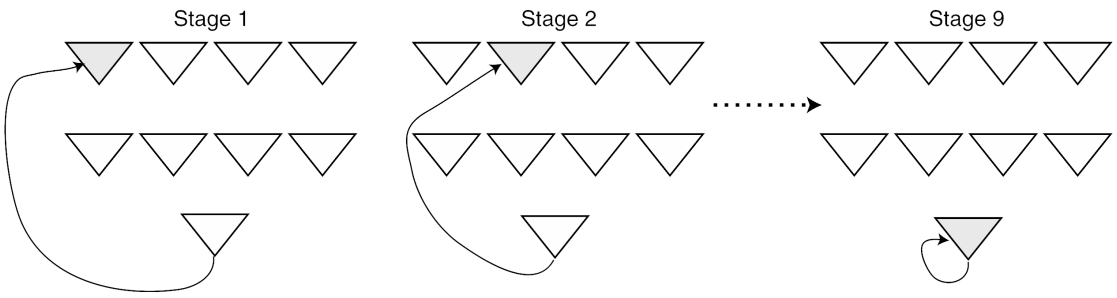


Figure 106: The 9 Stage Optimization Cycle

Variants of this scheme introduce greater flexibility in the choice of network nodes that a particular node is combined with in each stage. For example, in POOKA the best nodes from other populations are used while a particular node is evolved. This is called the greedy strategy in [73]. Another alternative is to randomly select the other nodes for a particular stage. In [75], more complex bookkeeping keeps track of *good* combinations of nodes and allocates fitness accordingly. It can be seen that the Optimization Cycle makes inefficient use of the fitness evaluator architecture. However, in terms of the EA it is possible that the Optimization Cycle enables a more efficient search of network configurations. This is explored further in experiments in Section 7.4.

7.3 Evaluation of POOKA Implementation

7.3.1 Resource Usage

The 9-node, 3-layer network was implemented at 50MHz. The resource estimates from both Synthesis and Place and Route software are summarized in Table 25. It can be seen that post synthesis the usage was estimated at 45%, while after place and route it grows to 64%. This indicates there is significant room to optimize the design. All components of the network and fitness evaluator architectures were designed with structural VHDL to which placement constraints can be applied. This effectively allows the design to be manually placed, which would bring the 64% usage closer to 45%. Manually placing the design would also allow higher clock rates to be achieved.

Resource	**Number**	**Percent of chip**
Post Synthesis		
Number of SLICES	8706 out of 19200	45%
Post Place and Route		
Number of SLICES	12427 out of 19200	64%
Number of BLOCKRAMs	56 out of 160	35%
Number of Tri-state buffers	2256 out of 19520	11%

Table 25: POOKA Resource Usage

Note also that even with row buffering POOKA only required 35% of the on-chip RAM resources available on the Virtex 2000E FPGA. This suggests larger spatial neighborhoods could be implemented and is discussed further in the Discussion in Chapter 8.

7.3.2 Evaluation of Speed-up

Evaluating speed-up of the CC implementation compared to software implementations is a difficult problem since raw processing speed is not the only factor. The quality of the feature extraction algorithm must also be compared. The network architecture presented is a unique and an entirely novel approach to AFE and therefore several comparisons are presented.

The first comparison compared the CC evaluation time to a software simulation of the POOKA network. The results are summarized in Table 26. The hardware evaluation times were averaged from runs involving 100 chromosomes evolving for 10 generations. Software evaluation times were averaged from runs involving 10 chromosomes, evolved for 2 generations on a 500 MHz Pentium III workstation. Both CC and software implementations were applied to training data sizes ranging from 65k-pixels through to 1M-pixels (the maximum size that can be loaded to the Firebird local memory).

Image Size (pixels)	Software Evaluation Time (Seconds)	RCC Evaluation Time (Seconds)	Speedup
65536	2.7	0.0016	1687
131072	5.4	0.0029	1862
262144	10.9	0.0055	1981
524288	21.7	0.0105	2066
1048576	43.2	0.0201	2149

Table 26: Evaluation Times for Software and RCC Implementations

The speed-up of the CC implementation over the software implementation is reported in the far right of Table 26. The large values obtained are due to the fact that the network architecture was designed particularly for implementation on CC. For example, the spatial operator uses a large number of *Arith-Morph-Mux* units that incorporate flexibility at a very low level. To implement equivalent behavior in software, many conditional statements must be included within nested loops. As a result, the software compiler has few optimizations available to it and the performance is poor. This is therefore an optimistic upper bound on speed-up.

A possibly more meaningful measure of performance can be estimated by considering a high-level approximation of network components. In this case, the quality of algorithm is not considered, but rather the execution time of a particular chromosome. For the software, execution time was estimated by implementing a number of optimized image processing operators. For each Spectral Processor in the network, a linear combination was used. For each Spatial Processor, a 5 by 5 neighborhood average was calculated. The software experiment therefore performed a total of 9 linear combinations of 4 images and 9 neighborhood averages. The execution times and relative speed-up are summarized in Table 27.

This software implementation is slightly simpler (and less powerful) than the CC implementation. This experiment can be considered to give a pessimistic lower bound on speed-up. Actual speed-up falls in between these two bounds, that is, in the range 100-2000 times a 500MHz Pentium III implementation.

Image Size (pixels)	Software Evaluation Time (Seconds)	RCC Evaluation Time (Seconds)	Speedup
65536	0.18	0.0016	112
131072	0.36	0.0029	124
262144	0.71	0.0055	129
524288	1.39	0.0105	122
1048576	2.75	0.0201	136

Table 27: Evaluation Times for Software and RCC Implementations

It can be seen that POOKA obtains a speed-up of two orders of magnitude compared to a software implementation of similar complexity running on a 500 MHz Pentium III workstation. This speedup illustrates the potential of using POOKA within a real-time AFE system. Note, the software implementation used in the comparison does not necessarily produce comparable AFE solutions. In Section 7.5, POOKA is compared to software AFE techniques that are generally more computationally intensive and therefore speedup would be expected to be larger for these cases.

7.4 Comparing EA Strategies

In this section, investigation is made of the POOKA EA strategy described in Section 7.2. In this experiment, three EA strategies are compared:

1. *Standard EA*: This is the strategy that was used in Chapter 5 and in experiments with the 2-layer network in Chapter 6. The entire network is represented by one chromosome (9 node chromosomes). Fitness is only calculated on the final output of the network and therefore, the Fitness Metric units associated with internal nodes were not used. Crossover in this case is similar to the strategy described in Chapter 6 for the two-layer network. In this case, a second crossover point is also selected in the second layer, and the same mechanism is used.
2. *Incremental EA*: This strategy was described in Section 7.2. There are 3 stages to the evolution, one for each network layer.
3. *Incremental EA + Optimization Cycle:* This is the extension of the Incremental EA, also described in Section 7.2. After Incremental Learning, each node is taken in turn and evolved for some time with fitness based on the final network output.

To compare the efficiency of the evolutionary search, the execution time for the 3 strategies was made approximately the same. The standard EA evolved a population of 200 networks for 120 generations. For the Incremental EA, evolution was for 60, 40 and 20 generations for the 1st, 2nd and 3rd layers respectively. For Incremental + Optimization, the 1st, 2nd and 3rd layers were evolved for 30, 20 and 10 generations respectively for Incremental learning. Optimization was then applied to each node in turn, and total optimization time is approximately 60 generations.

POOKA was applied to the Water, Golf and Urban feature finding problems introduced in Chapter 5. All 3 strategies were applied in 5 independent runs for each problem. The best, mean and standard deviation of the 5 scores produced by optimized networks are summarized in Table 24. The average execution time for each strategy is also reported in Table 24. It can be seen that the Incremental Learning strategies out performed the standard EA for the more difficult problems. For the remainder of the thesis, the Incremental Learning approach with optimization is used.

Standard: 50 s Perfect Classification: 1000						*Best*	*Mean*	*S.D.*
Water	1000	1000	1000	1000	999	1000	999.8	0.44
Golf	994	992	990	994	992	994	992.1	1.56
Urban	888	958	947	942	945	958	935.9	27.68
Incremental: 41 s								
Water	1000	1000	1000	1000	1000	1000	1000.0	0.04
Golf	994	993	991	992	990	994	991.9	1.43
Urban	968	958	961	960	965	968	962.4	4.04
Incremental + 1 Op Cycle: 38 s								
Water	1000	1000	1000	1000	1000	1000	1000.0	0.04
Golf	992	994	993	993	990	994	992.2	1.36
Urban	963	974	958	969	970	974	966.8	6.30

Table 28: Comparison of EA strategies

For comparison with the Morphological and Neural Network experiments of Chapter 5, POOKA was re-optimized and applied to the 3 test problems. In this case, a population of 100 was evolved for approximately 360 generations. Incremental Learning was applied for 120 generations, followed by 2 optimization cycles, equivalent to an additional 240 generations. This resulted in an execution time comparable to the time taken by Chapter 5 experiments. It can be expected that better performance could be achieved with a longer evolution. However, this experiment is concerned with comparing the POOKA architecture to the Morphological and Neural Network architectures that were explored in Chapter 5. A detailed investigation of convergence is made in Section 7.5.

The results on the training and test images are reported in Tables 29 through 31 for the water, golf and urban problems respectively. The fitness found at each node in the network is also shown to illustrate how successively higher scores are achieved with each layer. An effect of the optimization cycle is algorithm pruning. Nodes that do not affect the final network output are assigned a score of 500. Examples of this can be seen in Table 29.

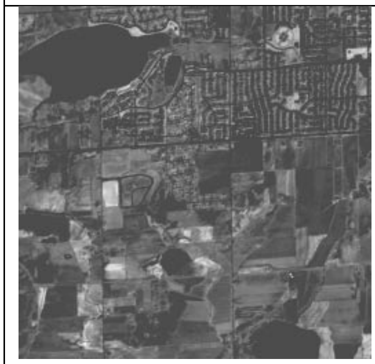
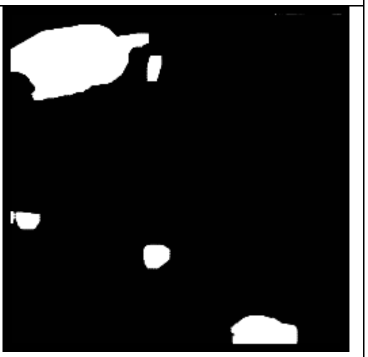
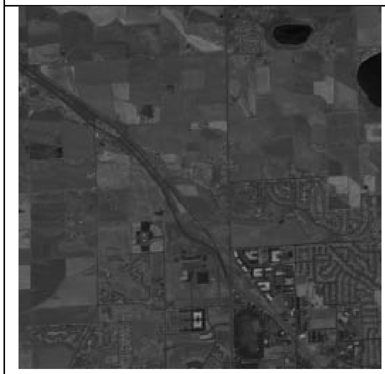
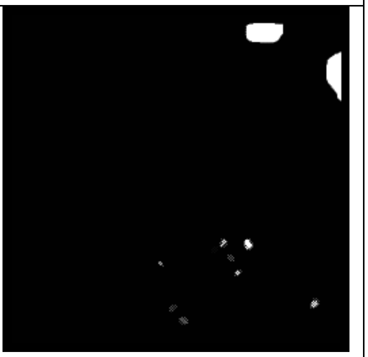
Water	1st Layer	2nd Layer	Output	Output Image
Training				
	500	596	1000	
	500	500		
	500	1000		
	1000	1000		
Testing				
	500	536	999	
	500	500		
	500	999		
	999	999		

Table 29: Results on Chapter 5 Water Problem

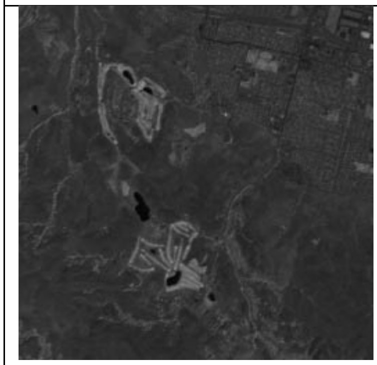
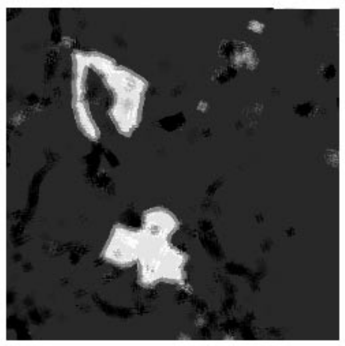
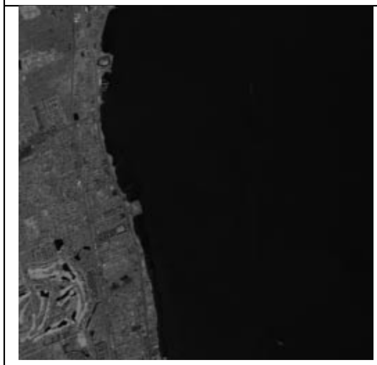
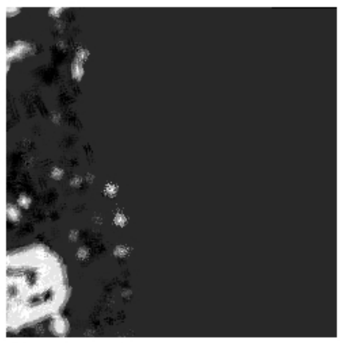
Golf Course	1st Layer	2nd Layer	Output	Output Image
Training				
	981	972	999	
	976	994		
	951	963		
	983	500		
Testing				
	947	972	994	
	960	993		
	750	737		
	970	500		

Table 30: Results on Chapter 5 Golf Problem

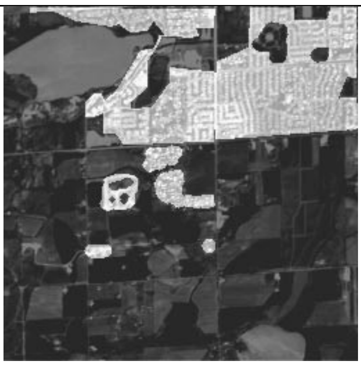
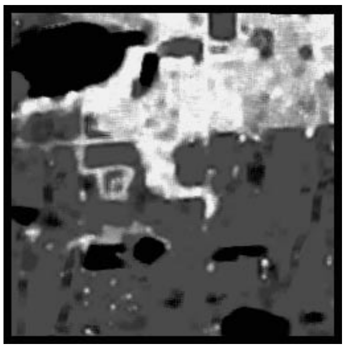
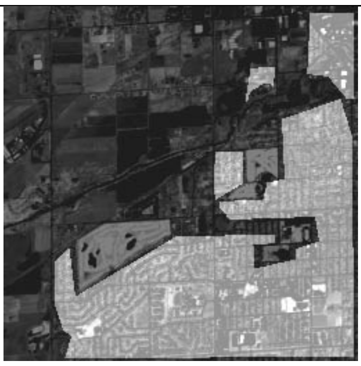
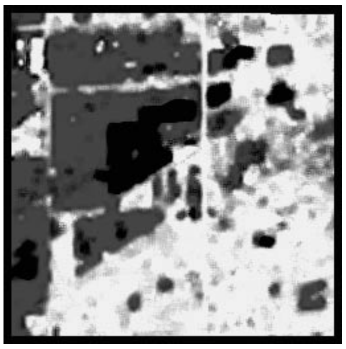
Urban Areas	1st Layer	2nd Layer	Output	Output Image
Training				
	614	537	979	
	656	962		
	943	950		
	500	500		
Testing				
	601	551	959	
	719	953		
	919	932		
	500	500		

Table 31: Results on Chapter 5 Urban Area Problem

7.4.1 Discussion

POOKA demonstrated promising results for the 3 test problems, particularly the urban area finder, where spectral networks of Chapter 5 had difficulty. Figure 107 illustrates the output images produced by each node when applied to the urban-area training image. Also included in this figure are some of the parameters found by the EA. For clarity, only a portion of the network configuration is shown. The 3rd node in the 1st layer implements a linear combination of morphological ring operators that seems to characterize the urban texture well. The output from this node is then enhanced in the 2nd layer. This can be seen in the output of the 2nd and 3rd nodes in the 2nd layer. These nodes apply functions of the 5 by 5 ring. The output from these nodes appears to be the main contributor to the output node. This node implements a function of the 3 by 3 ring.

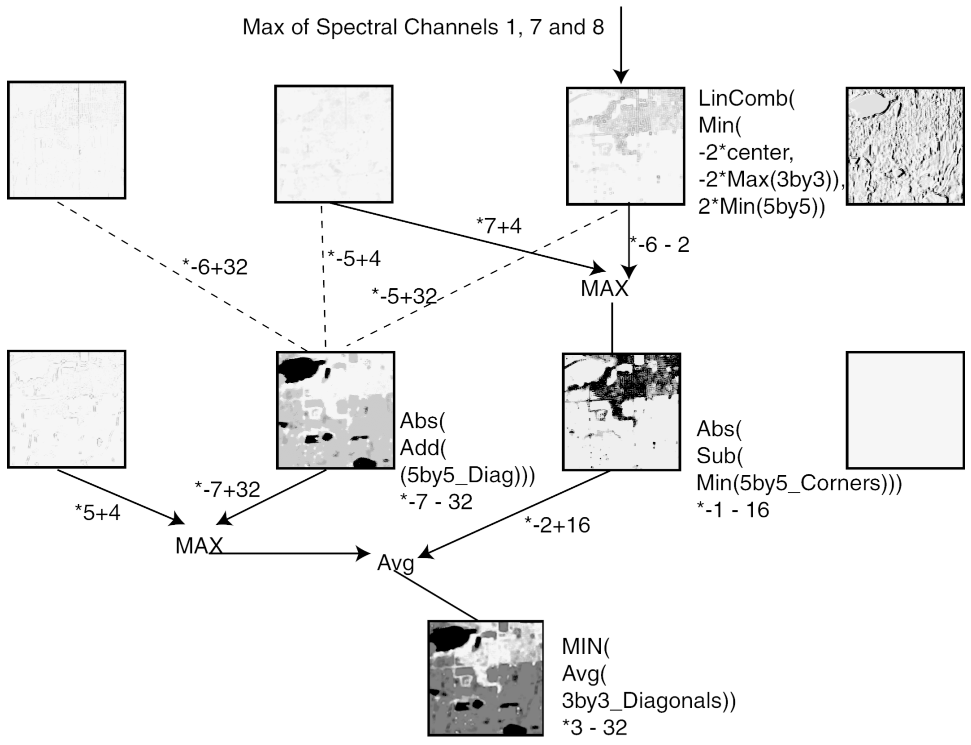


Figure 107: Urban Feature Finder Algorithm

Note that sometimes nodes are not used in producing the final network output. Examples of these can be seen on the right hand side of Figure 107. These nodes become apparent during the optimization phase of evolution. If the network score does not change, as different internal nodes are evaluated it is likely the node does not contribute. In POOKA, this results in a node with low fitness being used since the last node to be evaluated for a particular generation is often randomly generated. For example, it can be seen the far right nodes for the urban finder received a score of 500. This is useful for identifying non-contributing nodes without having to inspect the optimized chromosome.

7.5 AFE Quality Comparison

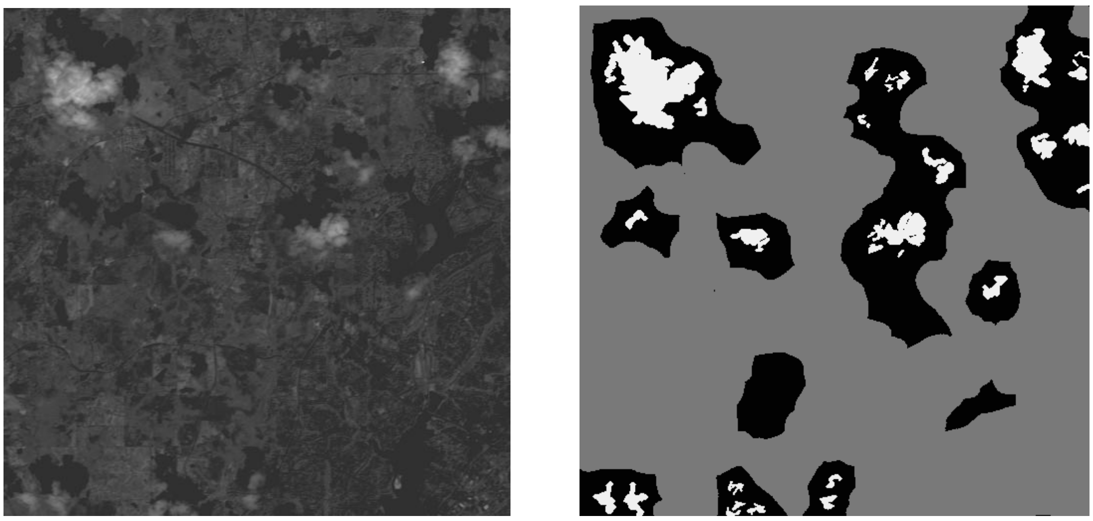
In this experiment POOKA is compared to software AFE algorithms using a test-set presented by Harvey in [142]. In this work a test set of 4 features was used to compare the GENIE system with more conventional remote-sensing classification algorithms. In this work, the conventional classifiers such as Maximum Likelihood and Spectral Angle Mapper, are applied only to the spectral dimension. The GENIE system used both spectral and spatial information. Both the GENIE system and spectral classifiers were described in the Literature Review.

Due to the stochastic nature of EA optimization, more robust performance can be expected by using multiple optimizations for a particular problem. In the comparison of [142], the GENIE system, which also uses an EA approach, is only applied once. Therefore, in the experiments to be described, the POOKA architecture is only optimized once for each problem. The EA schedule is similar to that used in the previous section and is summarized in Table 32.

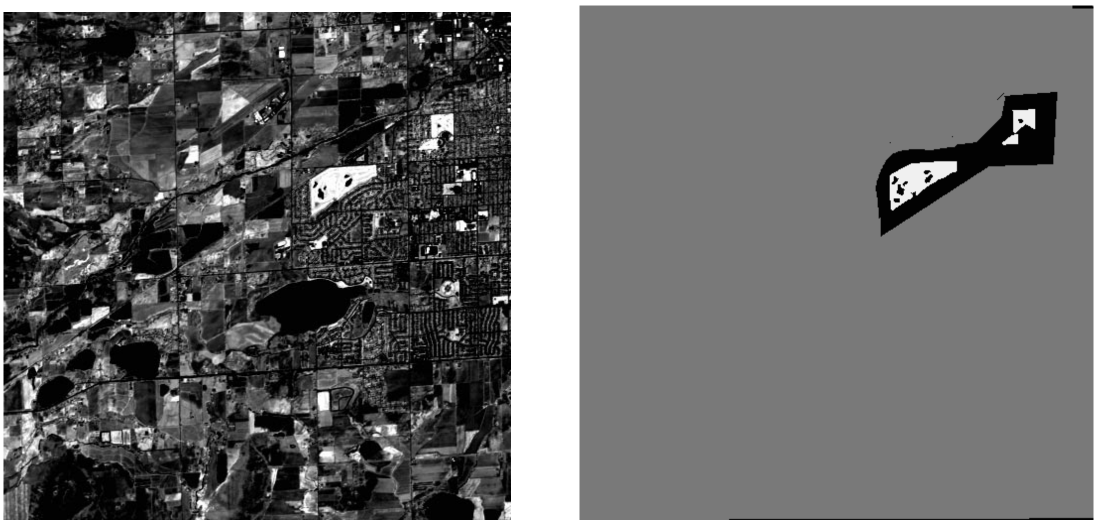
EA Parameter	**Number**
Population Size	100
Parents	80
Incremental Learning (generations)	120
2 Op Cycles (generations)	240
Reproduction	
Elite	10
Mutated Elite	20
Crossover	60
Mutation	90
Random Generation	10

Table 32: EA Schedule for AFE Comparison

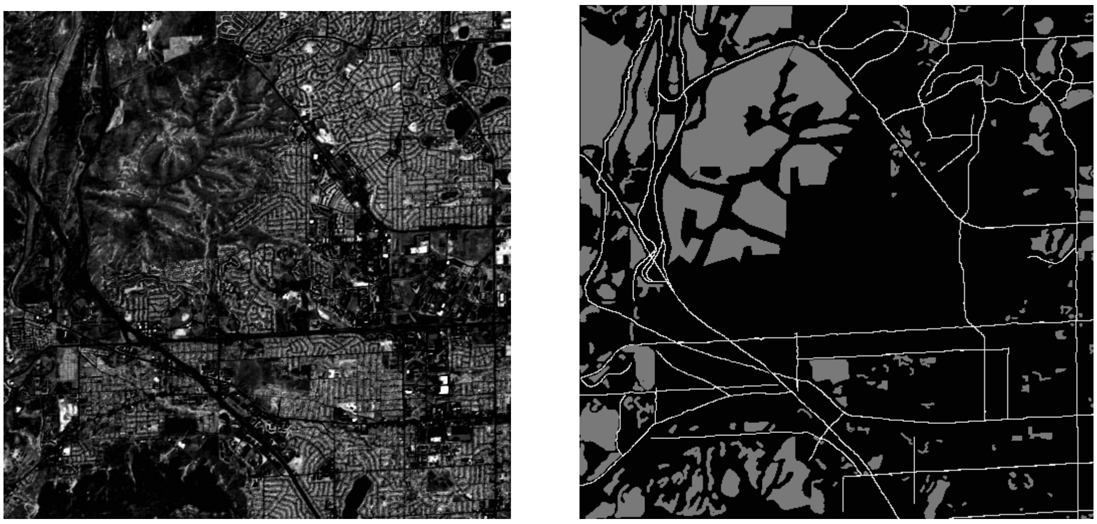
A target classification is specified for 3 different scenes for each of the four features. Figure 108 illustrates one of the three scenes, and corresponding target classification for each feature: A) clouds, B) golf courses, C) roads and D) urban areas. The images are from the simulated MTI data set [141], are 512 by 512 pixels in size and have 10 spectral channels.



A) Atlanta 1: Clouds



B) Denver 1: Golf Courses


C) Denver 7: Roads

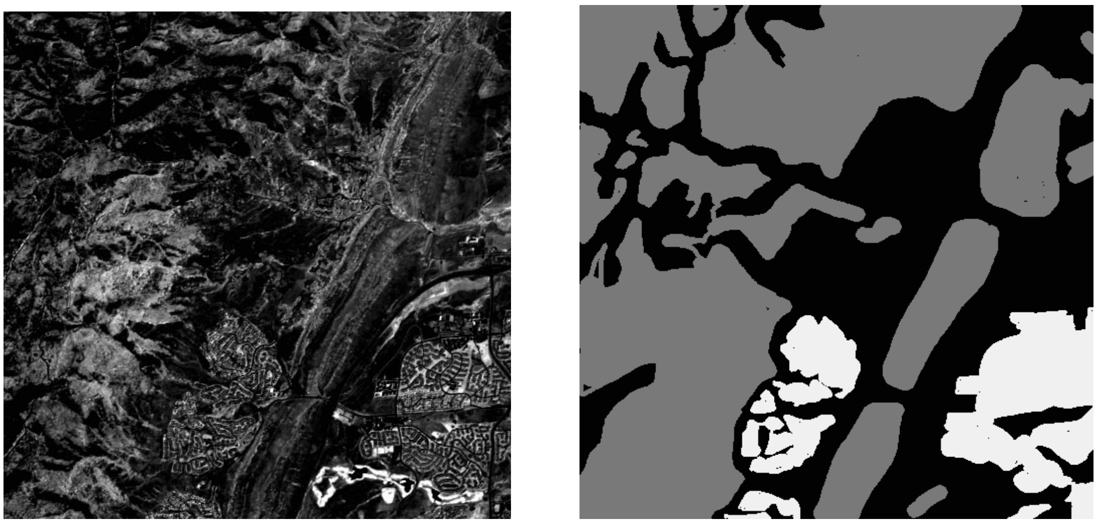

D) Denver 9: Urban

Figure 108: Example Training Images for A) Clouds, B) Golf Courses, C) Roads and D) Urban

POOKA is applied to all features, using each target classification in turn. The accuracy of the 12 optimized networks is summarized in Table 33. For each feature, a network trained on a particular scene is then applied to the remaining two scenes. The result on these out-of-sample target classifications is reported in Table 34.

Feature	Measure	Scene 1	Scene 2	Scene 3
Roads	Scene Name	Arm Site 6	Denver 7	Moffet Field 1
	Fitness	898	941	978
	Detection Rate	91.5%	94.7%	99.7%
	False Alarm Rate	12%	6.6%	4.2%
Urban	Scene Name	Denver 4	Denver 9	Moffet Field 1
	Fitness	989	977	984
	Detection Rate	99.2%	99.1%	98.7%
	False Alarm Rate	1.3%	3.6%	1.8%
Golf	Scene Name	Denver 1	Kennedy Space Center 2	Moffet Field 1
	Fitness	994	1000	994
	Detection Rate	99.9%	100%	100%
	False Alarm Rate	1.1%	0%	1.2%
Clouds	Scene Name	Atlanta 1	Atlanta 2	Clouds 1
	Fitness	999.9	999.8	999.9
	Detection Rate	100%	100%	99.9%
	False Alarm Rate	0%	0%	0%

Table 33: Results of POOKA optimization on Training Data

Feature	Measure	Scene 1 applied to Scene 2	Scene 1 applied to Scene 3	Scene 2 applied to Scene 1	Scene 2 applied to Scene 3	Scene 3 applied to Scene 1	Scene 3 applied to Scene 2
Roads	Fitness	669	786	585	868	639	751
	DR	98.6%	99.2%	91.7%	92.2%	39.9%	51.3%
	FA	64.9%	41.9%	74.6%	18.6%	12.2%	1.0%
Urban	Fitness	919	781	918	818	887	729
	DR	97.6%	59.7%	98.9%	99.9%	79.3%	46.9%
	FA	13.8%	3.6%	15.3%	36.4%	2.0%	1.1%
Golf	Fitness	500	508	548	517	782	500
	DR	0.0%	1.6%	100.0%	100.0%	100.0%	100.0%
	FA	0.0%	0.1%	90.3%	96.5%	43.5%	99.9%
Clouds	Fitness	997	993	999.8	987	995	993
	DR	99.5%	99.4%	100.0%	99.0%	99.9%	99.9%
	FA	0.1%	0.8%	0.0%	1.6%	1.0%	1.3%

Table 34: POOKA networks of Table 26 applied to Test Data

Figure 109 compares the POOKA results tabulated above with scores reported by several other classifiers reported in [142]. The other classifiers are GENIE, described in the Literature review, which uses a combination of spatial and spectral information. Results from Maximum Likelihood and Spectral Angle Mapper (SAM) are also included but these techniques were applied directly to the image cubes and therefore only spectral information is used.

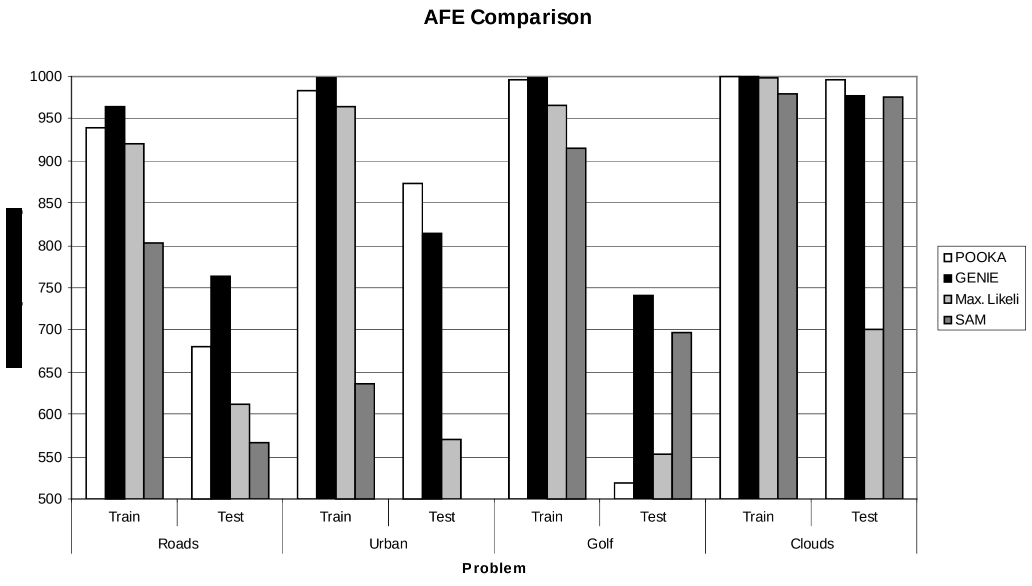


Figure 109: Comparison of POOKA performance to other classifiers.

7.5.1 Discussion

GENIE outperformed POOKA in almost all problems. This was not surprising due to the complexity of software algorithms available in the GENIE operator pool. For each problem, GENIE evolved 50 chromosomes for 1000 generations (unless a perfect classification of 1000 was reached) and took anywhere from 4 (when scores of 1000 were reached) to 12 hours to complete. In contrast, each POOKA optimization took 3 minutes and 40 seconds for each problem. It took 5 seconds to initialize the Firebird and load training images. A total of 214 seconds in evolution was made up of 73 seconds of incremental learning, and two 71-second optimization cycles.

In terms of generalization, the GENIE and POOKA approaches seemed to provide more robust solutions for out-of-training images. This was expected due to the combination of spectral and spatial processing available to these techniques. It can be seen that this was not true for POOKA on the golf course problem. POOKA had low classification error on the training images but very poor performance on the test images. While this result at first appears discouraging, it is suggested that this indicates a problem in the test set and not POOKA's generalization ability. Only 1 training image was used for optimization, and each system was only optimized once for each problem. Furthermore, each scene was taken from significantly different geographic locations over extended periods of time. While the ultimate goal is to develop algorithms that can perform well over the space of such conditions, it is generally accepted that training data should be representative of the problem space. It is therefore claimed that this test set makes it difficult to assess the generalization ability of any of the pattern recognition systems. In addition, the stochastic nature of EA means that some runs are better than others and is particularly true when the training data has been poorly sampled. This is supported by the fact that POOKA had excellent generalization performance on the smaller golf course problem in the previous section.

7.5.2 Additional Experiments

To make a more accurate assessment of POOKA's generalization ability, an experiment is formulated that attempts to provide a better sampling of the input distributions. In this experiment, a fourth scene is introduced for each problem. The 4 scenes are then divided into two images by tiling non-overlapping portions from each scene. This means that part of each of the 4 scenes is represented, leading to a more accurate sampling of the problem space. The tiled input images and training data for each of the four problems, clouds, golf courses, urban areas and roads are illustrated in Figures 110, 111, 112 and 113 respectively.

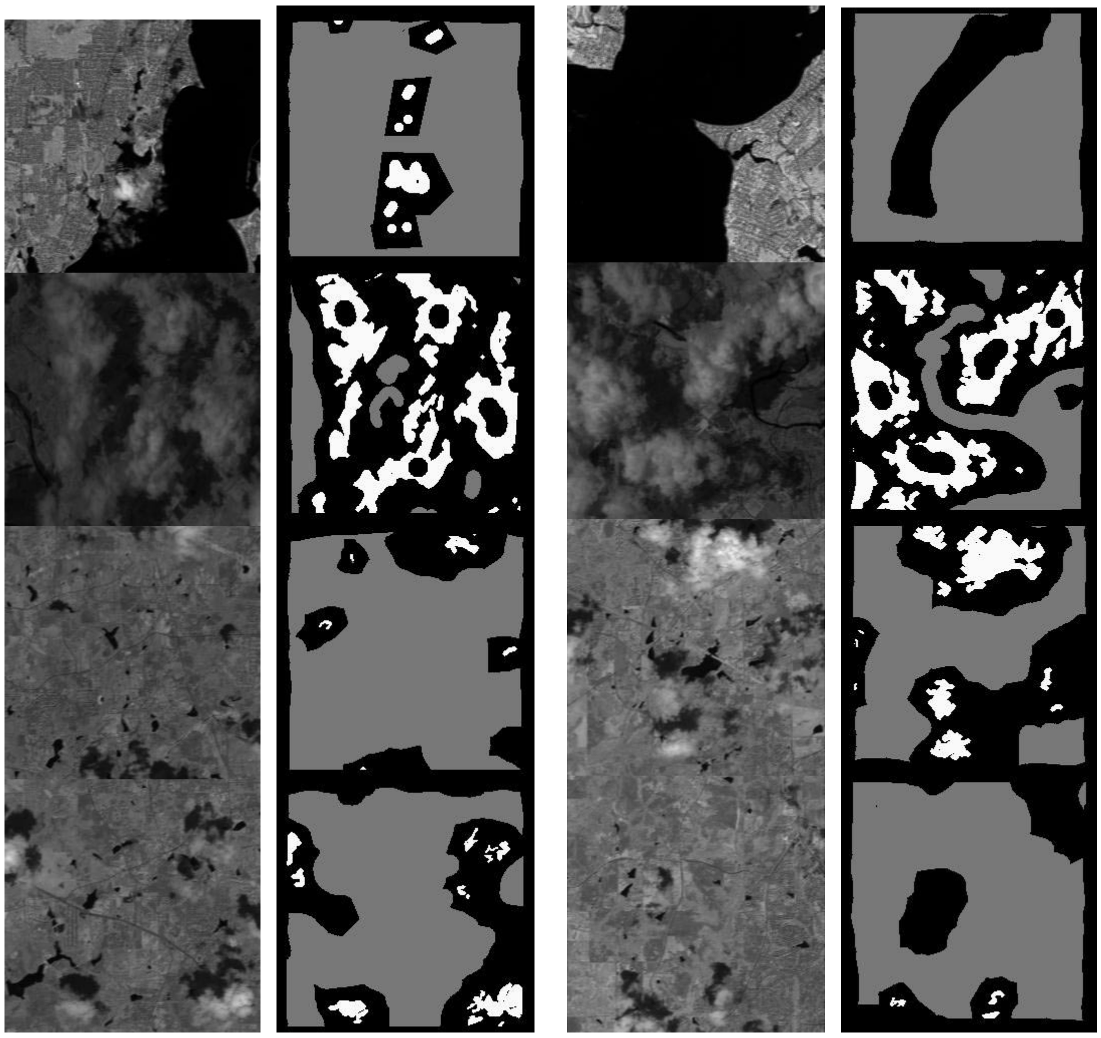


Figure 110: Cloud Problem (from left): Tiled image 1 and training data, Tiled image 2 and training data.

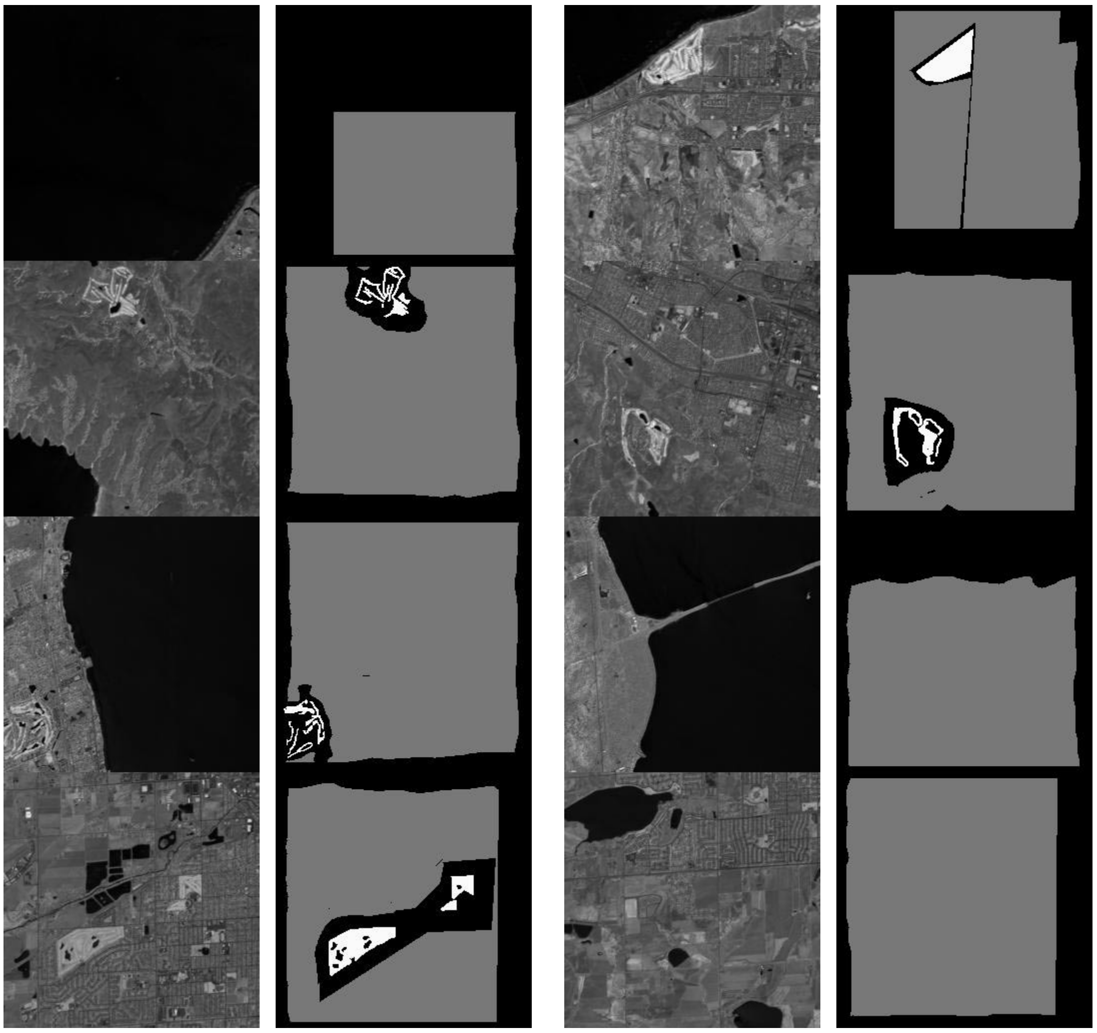


Figure 111: Golf Course Problem (from left): Tiled image 1 and training data, Tiled image 2 and training data.

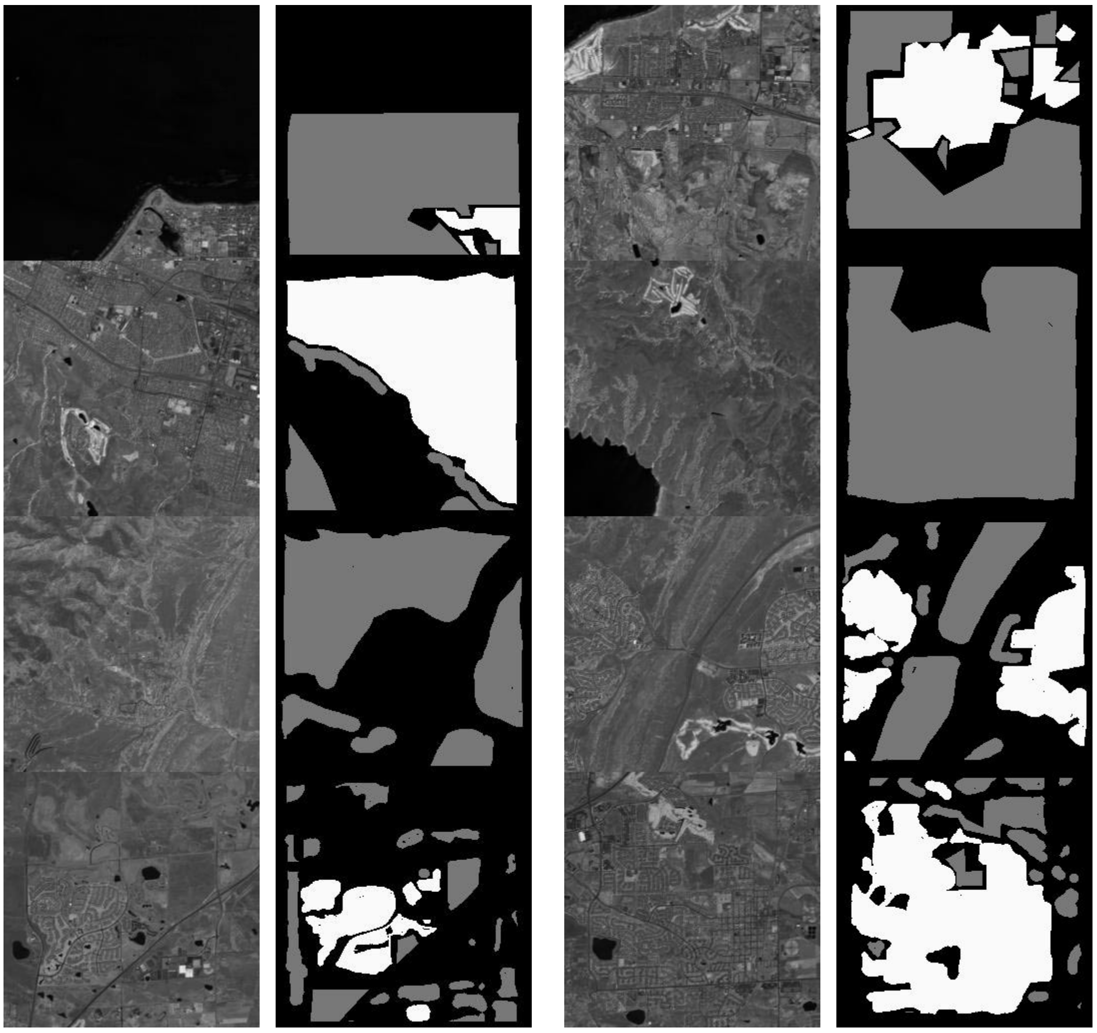


Figure 112: Urban Area Problem (from left): Tiled image 1 and training data, Tiled image 2 and training data.

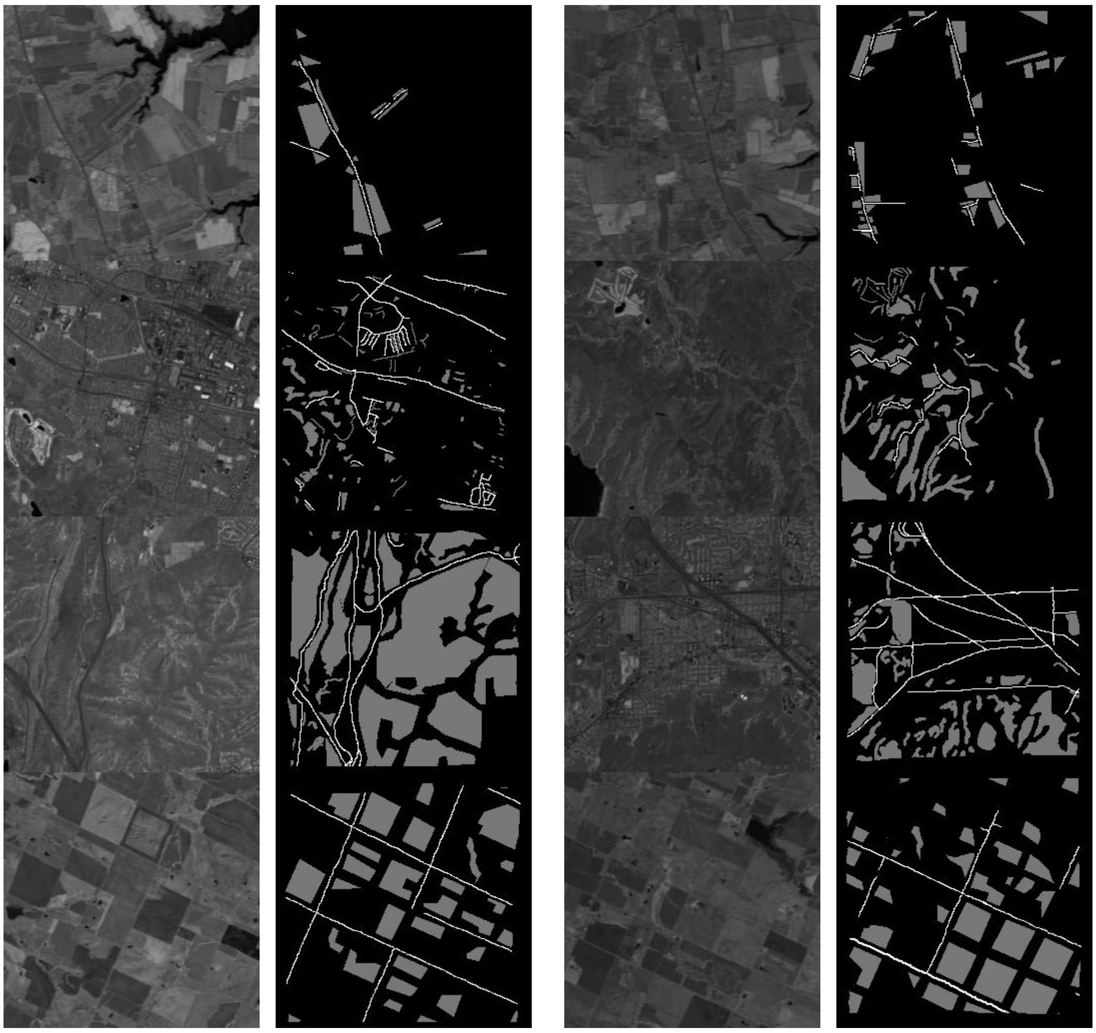


Figure 113: Road Problem (from left): Tiled image 1 and training data, Tiled image 2 and training data.

For this experiment a comparison is made between three pattern recognition systems. The first is the GENIE system, described previously, which evolves a population of 100 chromosomes for 100 generations. The second system, known as AFREET, is based on a recently introduced class of pattern recognizers known as Support Vector Machines (SVM). SVM have been developed from recent advances in statistical learning theory and have gained considerable interest for pattern recognition problems since they have guaranteed bounds on generalization error. The AFREET system has explicit measures to help generalization, and therefore is an excellent candidate with which to compare generalization performance. A detailed description of these explicit measures is beyond the scope of this thesis, but readers are referred to Vapnik's pioneering work on the topic [143]. Similar to GENIE, AFREET is a hybrid architecture system with a stochastic feature selection stage, followed by a SVM classifier [144]. The feature selection stage contains a rich variety of both spatial and spectral algorithms similar to the GENIE operator pool.

Convergence of EA optimization techniques is difficult to define. For this reason POOKA was applied to all problems at 4 different levels of effort, which are detailed in Table 35. However the question arises, can better performance be expected with even greater effort? Fitness verse generation plots can provide subjective answers to this question. Figure 114 presents this measure for the Road 1 problem at extreme effort. Low, Medium and High levels of effort reach generation 118, 360, and 840 respectively. It is suggested that with a high to extreme level of effort, the POOKA optimization procedure can be considered to have converged.

Effort Level	**Low**	**Medium**	**High**	**Extreme**
Execution Time	34.7 seconds	3.2 minutes	15 minutes	56 minutes
Population	50	100	200	200
Generations	120	240	240	480
Optimization Cycles	1	2	6	12

Table 35: Levels of effort in application of the POOKA system.

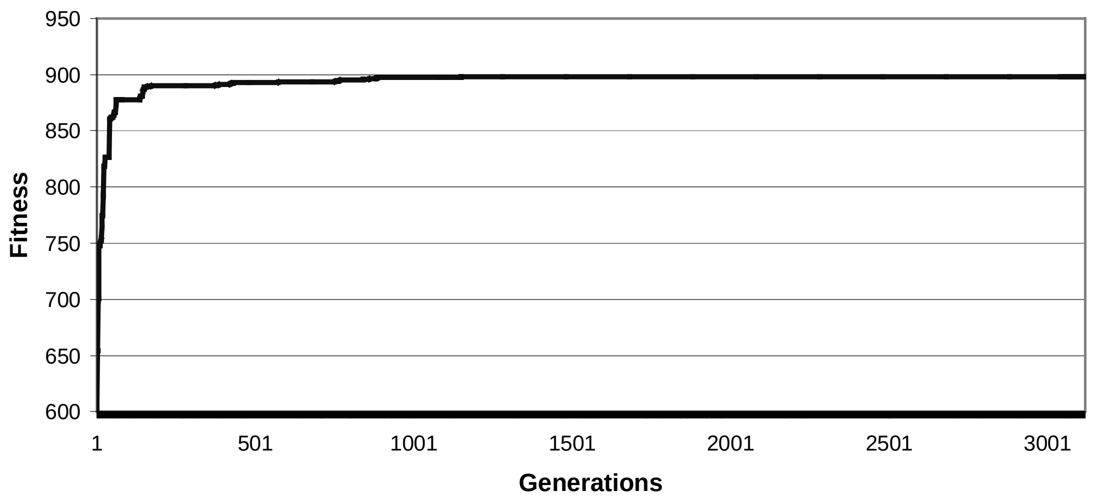


Figure 114: Fitness verse generations for the Road 1 problem at extreme effort.

All systems were trained on each tiled image, for each problem, in turn. The result is then applied to the second image for each problem to obtain an out-of-sample test score. The results on the training images are summarized in Table 36. The average training time for the GENIE system was 19 hours. The AFREET system sub-samples the training data prior to optimization. For this reason, execution time does not depend on the training image size, as in the other systems, but rather the problem difficulty. Execution times varied from 9 minutes through to 50 minutes and averaged 19 minutes for all the problems. The performance on the test images is summarized in Table 37.

Training	POOKA				GENIE	AFREET	
Image	Low	Medium	High	Extreme	(Avg 19 hrs)		Time (minutes)
Cloud 1	997.6	999.4	1000	1000	999.9	995.7	10.3
Cloud 2	994	994.9	999.8	999.5	998.6	998.7	9
Golf 1	992.3	995	998.8	997.2	997.3	999.7	9.3
Golf 2	996.1	998.7	999	999.7	998.7	997.9	13.8
Urban 1	881.6	974.3	983.9	987	992.1	984.7	25.3
Urban 2	947	982.6	991.3	993.2	996.5	992	12.3
Road 1	818.3	878.8	892.6	898.2	911	903.6	50
Road 2	904.1	930	917.6	925.4	944.6	935.9	22.7

Table 36: Fitness Scores achieved on Training Data

Training	POOKA				GENIE	AFREET
Image	Low	Medium	High	Extreme		
Cloud 1	989	968	981.4	991	978.5	819.6
Cloud 2	995	995	997.2	997.3	999.9	971.3
Golf 1	836.5	962	987	971.4	823.5	966.7
Golf 2	977.6	984	986	987	968.6	998.7
Urban 1	852	957.5	948	947.4	973.2	980
Urban 2	817.2	858.3	922.5	946	936.2	943.5
Road 1	837.4	897.3	883.8	909.3	935.5	913.7
Road 2	800.8	855.9	847.9	816.6	869.7	838.2

Table 37: Fitness Scores achieved on Testing Data

The results of Tables 36 and 37 are summarized in Figures 115 and 116. For each problem, the two training scores are averaged to produce a single figure for the problem. Similarly for Figure 116, the two test image scores are averaged to produce a single result. In addition, Table 38 shows the training and testing scores that were achieved when the POOKA system is applied with 8 different random seeds to the Road 1 problem. This experiment was conducted to investigate the variability of the POOKA system at the different levels of effort.

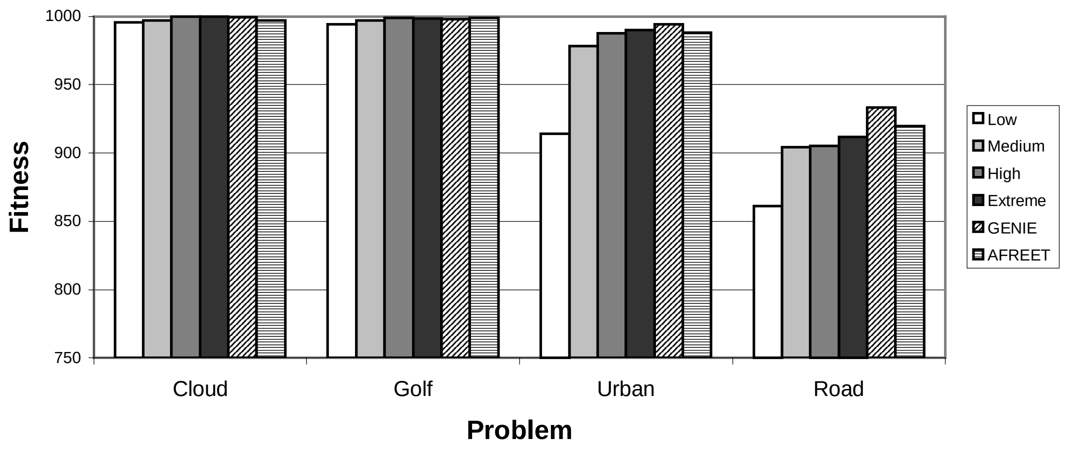


Figure 115: Summarized results for training images

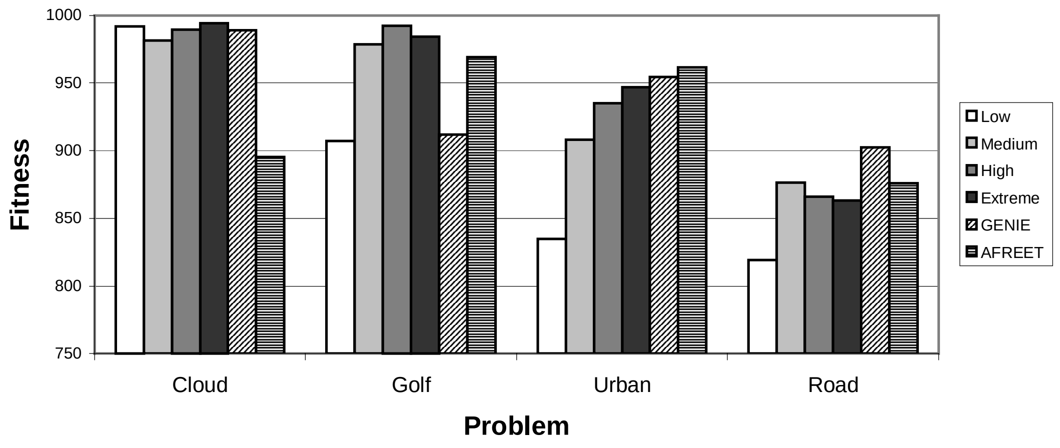


Figure 116: Summarized results for test images.

Run	1	2	3	4	5	6	7	8	Average	S.D.
Low										
Training	785.6	835.2	837	813.5	793.4	802.6	845.8	774.7	811	26.2
Testing	798.3	866.4	827.2	827.2	816.5	796	893	790.5	826.9	36.1
Medium										
Training	878.5	904.8	901.7	859.4	912.7	880	831.8	872.3	880.2	26.6
Testing	896.7	858	894.2	890	910.9	903.7	876.3	890.3	890	16.5
High										
Training	902	908.9	873.6	909.6	883.3	902.3	901.6	884.2	895.7	13.4
Testing	901.6	911.1	899.4	928.5	906.9	895.2	916.6	865	903	18.7
Extreme										
Training	894.1	922	899	897.9	871.9	925.4	902	903.7	902	16.7
Testing	871.5	917.1	914.4	909.6	896	913.2	911.1	916.3	906.2	15.5

Table 38: Fitness Scores over 8 runs for the Road 1 problem (Tested on Road 2).

7.5.3 Further Discussion

Overall, the POOKA results are promising. With low effort, the POOKA performance was poor on both training and test images. This is indicative of the increased difficulty of the problems, and therefore does not measure the true capacity of the POOKA system. With increased effort, the POOKA system shows potential as a practical pattern recognition system. Test data results indicate that POOKA has good generalization ability. This is particularly evident in the simpler cloud and golf course problems, where POOKA appears not to over-fit, even when an extreme amount of effort is applied. Also, for most of the problems, a higher accuracy on the training data has resulted in a higher accuracy on the test set.

For the more difficult problems, POOKA is usually outperformed on the training data by the GENIE and AFREET systems. This indicates that POOKA may lack classification power for difficult training data, compared to the software systems. POOKA has a fixed number of nodes with which it can work, and therefore this is not surprising. In contrast, both GENIE and AFREET systems are able to form extremely complex algorithms to fit training data. It is hypothesized that the limited resources of the POOKA system may be responsible for its good performance on test data, however it does limit the application of POOKA to more difficult problems.

The results of Table 38 indicate there is variation in performance from one run to the next, particularly at low levels of effort. While this is expected from an evolutionary algorithm system, an interesting direction for future research would be to find EA strategies that can reduce this variation. It can be seen for high-levels of effort the variation is reduced, which indicates that a more efficient EA search strategy may help with this problem. In addition, it is noted that for two of the runs at extreme level of effort, POOKA actually obtained higher training data scores than both the GENIE and AFREET systems. While this indicates that POOKA has potentially sufficient classification power, the difficulty in obtaining these scores is a problem. A more efficient EA strategy would help with this problem. Another solution would be to increase the classification power of the system, hence providing a richer solution space that could potentially be searched more easily.

To improve the POOKA system, it is concluded that the classification power should be increased. The fact that the system has a fixed set of resources for problems of varying levels of difficulty may also be a limitation, and a more flexible use of the POOKA resources may be appropriate. Ways of increasing the classification power of the POOKA system are described in Chapter 8. At the same time, it is known that increasing classifier complexity can lead to problems of over-fitting. Therefore, it is suggested that including explicit measures to help generalization would be beneficial in a more complex POOKA system. This is a topic of future research, but some directions are also included in Chapter 8.

7.6 Parallel Evolution of Multiple Features

In this section an interesting application, made possible by the multiple fitness metrics is described. Section 7.1 briefly described how the top-level POOKA architecture was extended to load 9 target classifications into on-board memory. These 9 target classifications are supplied independently to the 9 Fitness Metric units. Up to this point, only one target classification has been used and therefore 9 duplicate copies of the target classification have been stored. This flexibility in target classifications can be used to optimize different nodes for different features of interest. This can be useful in exploiting relationships between high-level features in an image. For example, if the feature of interest is beach, it is possible that first finding the water in the image is useful.

The beach example is used to demonstrate the principle. The training data is a 3-channel image, from the KSC instrument. Two networks are optimized. For the first network, all nodes were optimized using the beach target classification illustrated in Figure 117B. For the second network, 2 target classifications were supplied. The 1st and 4th nodes in the first layer were optimized using the water target classification of Figure 117C. The remaining nodes were optimized for the beach target.

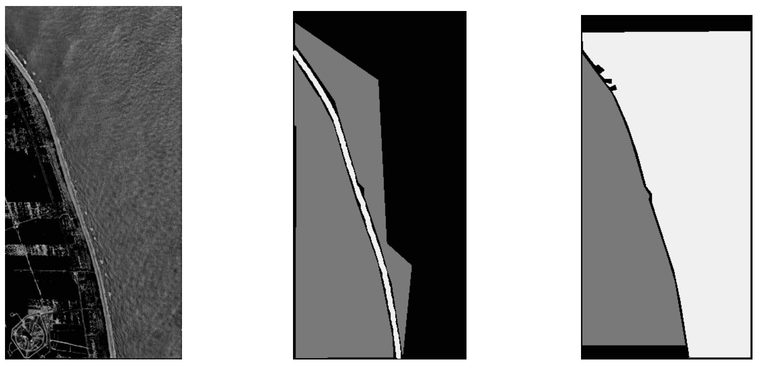


Figure 117: A) Original Image B) Beach target and C) Water target

The node outputs from the two optimized networks are illustrated in Figure 118A and B respectively. One run was used to optimize both networks, and a better result was obtained by using the multiple target classifications. Inspection of the optimized chromosome also reveals that the water classification of Node 4 in the 1st layer was indeed used in subsequent layers to produce the final network output. Although it is likely that a network optimized with just the beach classification can solve the problem just as well, this outcome demonstrates the potential of the approach.

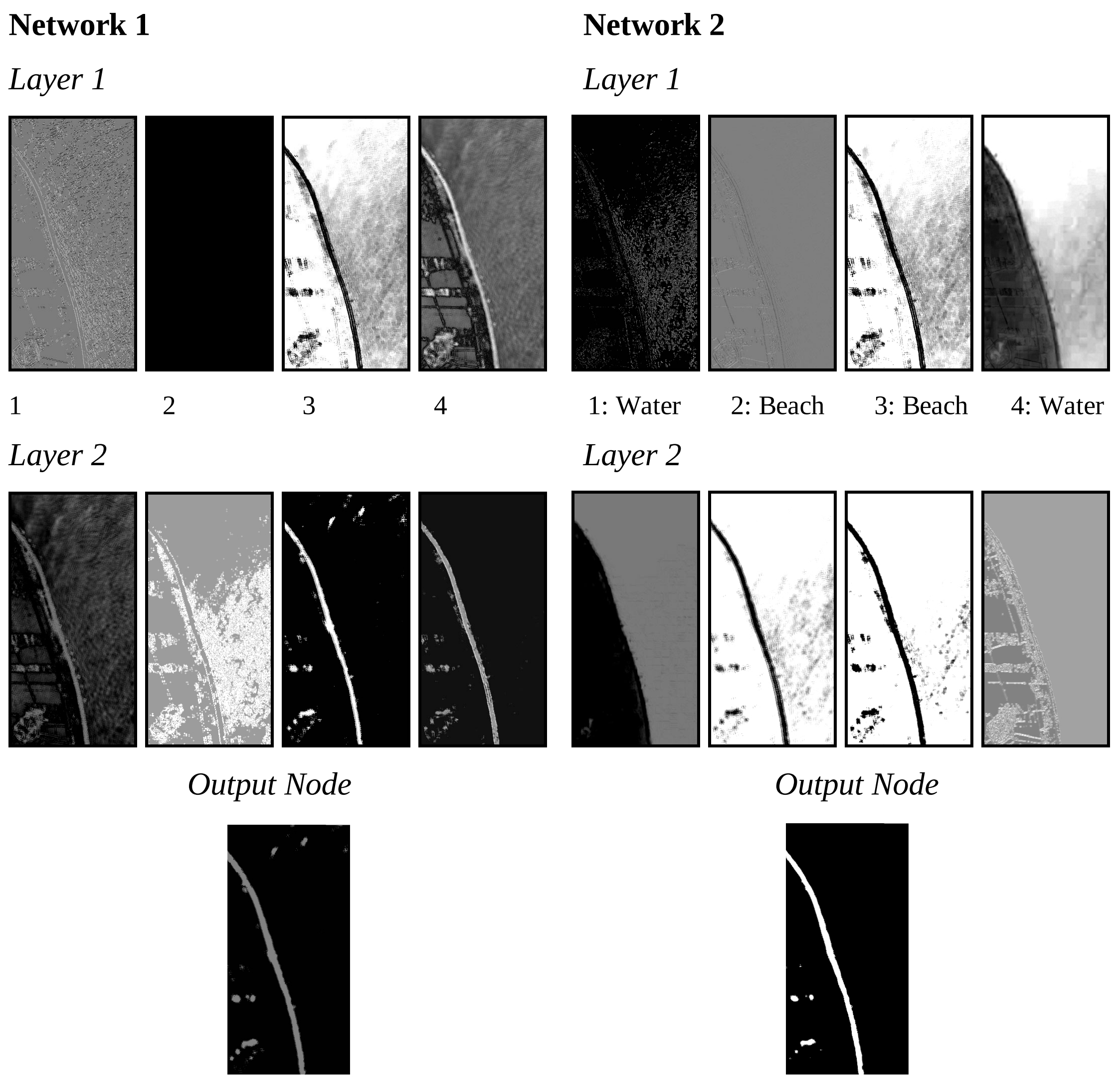


Figure 118: A) Single beach classification and B) Multiple target classifications

It is possible, that the multiple target classification technique could be of potential benefit in more subtle ways. For example, it is often desirable to make a complete scene classification where every pixel in the image is classified assigned to one of several classes. If a network has sufficient nodes, evolving feature finders for these classes in parallel may be better than developing feature finders independently. This technique could potentially benefit from a software implementation, where variable numbers of nodes or sub-networks could be dynamically allocated to features depending on classification difficulty. However, ways of incorporating such flexibility by using flexible tri-state routing of target classifications can be imagined.

7.7 Chapter Summary

This chapter has put many of the ideas developed in previous chapters to test. A 3-layer network of multi-spectral processing nodes was implemented. This network used approximately 65% of the Firebird resources, and was applied to image data sets with two orders of magnitude speedup compared to software of similar complexity.

A comparison of AFE accuracy was made with several software techniques. POOKA AFE showed promising results compared to the spatio-spectral GENIE software system, with the advantage of reasonable optimization times, and at least two-orders of magnitude speed-up in data throughput. The novel contributions of this chapter are:

- A Fitness Evaluator implementation using distributed fitness metrics that allows independent populations of network nodes to be optimized in parallel.
- A comparison of EA strategies in terms of search efficiency in CC. The EA strategies included standard EA, incremental learning, and a co-evolution type optimization cycle.
- A comparison was made between the POOKA approach and other classification techniques. Results suggest increasing the capacity of the POOKA system may be necessary for more difficult problems.
- Hardware accelerated evolution of multiple features in parallel was used to leverage high-level information.

Chapter 8

Discussion

8.1 Summary of Contributions

This thesis has presented a new approach to hardware AFE using a combination of Evolutionary Algorithms and Custom Computers. Throughout the course of the thesis, novel contributions have been described at the end of each chapter in Chapter Summaries. The following is an abbreviated list of the more significant contributions made by each chapter.

Chapter 3

The Generalized Chromosome and Hardware Reuse

Chapter 3 presented design choices and considerations useful to a wide range of EA implementations using Custom Computers. The idea of *Generalized Chromosomes* and methods of hardware reuse are believed particularly useful to modern FPGA devices where Rapid Reconfigurability is not available.

Chapter 4

Novel Maximally Parallel Architectures

Chapter 4 presented several algorithmic variants that can be efficiently implemented on FPGAs. Algorithm decomposition of Cellular Automata, led to the *hybrid* XC6216 CA model, which could be efficiently implemented and also allowed problem specific constraints to be easily applied. Novel thresholding strategies for stack filters were also suggested and explored.

Chapter 5

Comparison of Neural Network and Morphological Networks

Chapter 5 has made one of the first objective comparisons of morphological and neural networks for solving practical problems. Comparisons were made both in terms of the hardware resources required for implementation as well as the quality of classification algorithm.

Chapter 6

A Novel Multi-Spectral Network Node

- A self contained, modular network node capable of both feature extraction and classification was developed for multi-spectral image processing.
- The use of hybrid *morphological-linear* building blocks enabled the implementation of many traditional image-processing algorithms and classification techniques.
- The well-motivated arrangement of these building blocks was able to implement a rich variety of hybrid linear/nonlinear spatial filters.

Chapter 7

POOKA Implementation and Experiments

- The implementation of distributed Fitness Metrics improved the efficiency of search for the evolutionary network architectures in hardware.
- A comparison in terms of search efficiency was made between Incremental Learning and Co-evolutionary (Optimization cycles) EA strategies.
- A comparison was also made of POOKA to advanced spatio-spectral software solutions and more traditional spectral classification techniques.
- The use of POOKA for evolving multiple features in parallel was demonstrated.

8.2 Towards POOKA II

In comparison to other techniques, POOKA results were encouraging, but they also indicated there is room for improvement. An optimized POOKA implementation presented in Chapter 7 requires approximately 45% of the resources available on the

Firebird CC. Several other Custom Computers are being produced with even greater computational resources. This section proposes additions and extensions to the POOKA architecture that will require additional resources, but may lead to improved performance.

The experiments of Chapter 7 led to the conclusion that the POOKA system would benefit from increased classification power. Sections 8.21 through 8.24 suggest mechanisms by which this could be increased within the context of the POOKA architecture. At the same time, it is well known that increasing the complexity of the classifier can also lead to the problem of over-fitting. It is likely that with a substantial increase in classifier capacity, explicit measures to control generalization would be required. Specific steps towards this goal are discussed in Sections 8.2.5 through 8.2.7.

8.2.1 Multi-Spectral Data Sets

The fundamental aspect of multi-spectral data sets addressed in this thesis was the combination of spectral and spatial processing. However, the field of remote sensing defines a large number of algorithms that could also be incorporated to improve performance on particular problems.

One operation that was described in the Literature Review, and considered particularly useful is the band ratio. It is possible to include this operation within the node itself with a generalized arithmetic pipeline described in Chapter 3. However, this building block is substantially larger than others used within the node and therefore band ratios may be better implemented as preprocessors to the network.

Another consideration in practical AFE that has not been addressed in this thesis is spatial resolution. It is possible that POOKA will be applied to a wide variety of data sets, each with different spatial resolutions that may, or may not, be well suited to the 5 by 5 neighborhood operations that are implemented. One solution would be to perform image re-sampling in software to produce a standard resolution prior to loading training images. Another solution would be to dynamically resample spectral channels from one chromosome to the next. This is particularly attractive for developing algorithms that use data from multiple sensors and therefore use multiple training images with different spatial resolutions.

A flexible preprocessor to POOKA input nodes is therefore thought desirable. This preprocessor could be included in the network chromosome and then optimize band ratios and spatial resolutions on a problem-by-problem basis.

8.2.2 Bit Widths

Improved performance could be expected by increasing the bit-widths, and therefore the accuracy of network nodes. In this thesis, 8 bit data paths were used and although this was sufficient to demonstrate the potential of the approach, larger bit-widths may lead to improved performance. In some optimizations, POOKA found solutions where 1st layer network nodes produced almost binary outputs. This is partly due to the limited dynamic range available to network nodes. Raw data from current multi-spectral sensors is often fixed-point integers that use 12 to 14 bits. By using 8-bit data paths, information is therefore lost before any processing occurs. Latest CCs have large computational resources and these continue to grow. Larger bit-width could therefore be implemented with the present design.

8.2.3 Extending the Spatial Processor

The Spatial Processor described in Chapter 6 was shown to include a variety of potential useful spatial filters. The Spatial Processor was also shown to produce rotationally invariant spatial filters by constraining the configuration to one quadrant. While this rotational invariance is desirable, more powerful spatial filters could be implemented by allowing asymmetric spatial functions. Examples of this, described in the Literature Review, include Laws convolution masks, Gabor filters and linear morphological structuring elements. With these approaches a bank of filters, each representing a specific orientation, were used to achieve rotational invariance.

In a similar way, the Spatial Processor could be enhanced to include multiple rotated versions of an asymmetric spatial function. In this case, all 4 quadrants of the Spatial Processor would be included in the chromosome. Four Spatial Processors would then be implemented in parallel, each using a different orientation of the configuration. This is illustrated in Figure 119. ML represents the hybrid Morphological-Linear building blocks. The Spectral Processor has been decomposed into the two-layer network of these building blocks. It can be seen that they are also used to combine the multiple rotations of the Spatial Processor into a single output.

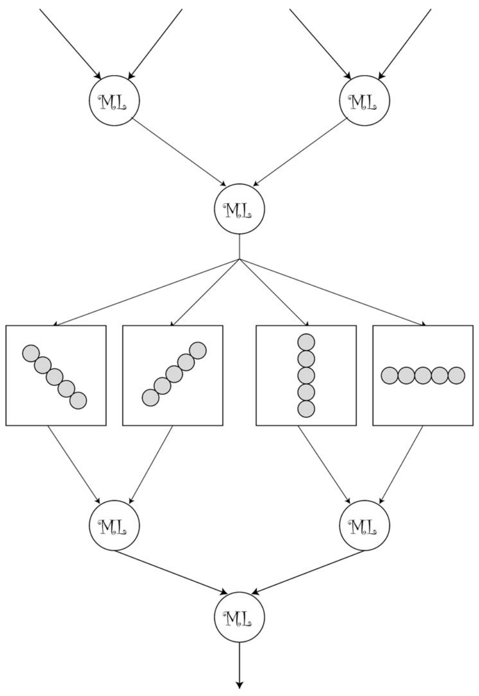


Figure 119: Revised Spatial Processor

By increasing the computational complexity in the spatial domain, smaller band filters are implemented in the frequency domain. This would improve the Spatial Processors ability to discriminate between textures. The large amount of on-chip RAM available in latest FPGAs such as the Virtex 2000E, means this approach could be incorporated in the current design.

8.2.4 Nodes as Functions of State

Another interesting direction for future work is inspired by the work in Chapter 4. The primary difference between the cellular architectures and the POOKA network architecture is state. It is possible to extend the POOKA top-level architecture to include state. Implementation is made simpler by the fact that node outputs are already sent to the local memory of the Firebird CC. These node outputs can be considered the current state of the network, and therefore returned as input to the network to implement functions of state. This is illustrated in Figure 120.

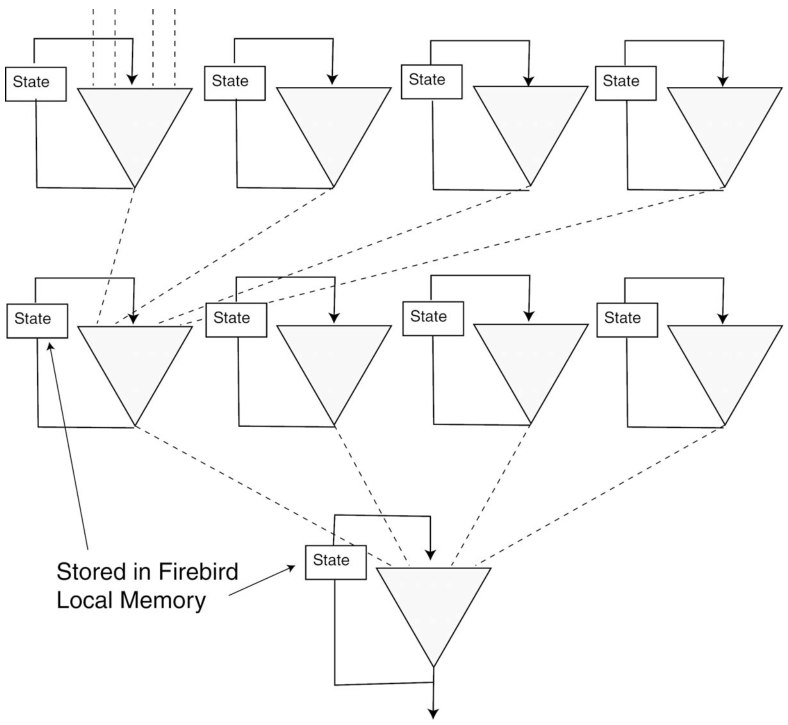


Figure 120: A Cellular Architecture Interpretation

This implementation has several interpretations. In one sense, the network would implement multi-layered cellular automata, where the rule set is a particular subset drawn from image processing. In another sense, it would be similar to architectures seen in the field of Cellular Neural Networks [145]. More accurately, POOKA would implement a multi-layered Cellular Morpho-Linear Network.

In this implementation, a network evaluation would require multiple passes through the image. A potential application of this seemingly computationally intensive extension is real-time image processing. Since network nodes are a function of state, information contained in the time domain could be exploited. This type of information is essential for applications such as change detection.

8.2.5 Fitness Metrics

This thesis used the weighted hamming fitness metric due to its simplicity of implementation. With this metric, only the sign of output pixels contribute to the score and therefore classification error could be accumulated with single bit counters. In theory, a better fitness metric would include some measure of how close output pixels are to the decision boundary. An example of this fitness metric is shown in Equation 17. This equation will produce fitness scores in the same range as the hamming metric used in this thesis, that is, 1000 is a perfect classification.

$$Fitness = 500 * \left(1 - \frac{\sum_{n=1}^{T} (P_{Max} - P_n^t)}{T * P_{Max}} \right) + 500 * \left(1 - \frac{\sum_{n=1}^{F} (P_{Min} - P_n^f)}{F * P_{Min}} \right) \qquad \textbf{(17)}$$

In Equation 17, P_{Max} and P_{Min} correspond to the maximum and minimum of the dynamic range, P_n^t and P_n^f correspond to the nth true and false pixel value respectively and T and F are the total number of true and false pixels. With 8-bit data paths, the perfect score (1000) would correspond to all true pixels having a value of 127, and all false pixels having a value of –127.

This fitness metric is more expensive to implement on FPGAs than the hamming metric since the difference between the pixel value and the target values must be accumulated. The accumulator bit-width will be larger than for the hamming metric. However, this extra cost is well within current FPGA device capacities, and better results, particularly in terms of generalization, may be achieved.

Another measure of fitness is inspired by the Stack Filter experiments in Chapter 4. The novel median thresholding strategy suggests that pixels could be assigned to classes by comparing the values from two output nodes. This technique is seen for two class problems in neural networks. In this case, each node is associated with a class. The output node is the posterior probability of class membership, and therefore the pixel is assigned to the class whose node has the largest value. Based on results in Chapter 4, this could potentially improve performance by producing a cleaner, or smoothed, output image. This would be at the cost of implementing an additional node.

8.2.6 Cross-Validation

If, after increasing the classification capacity, the POOKA system appears to over-fit training data, cross-validation could also be used to explicitly improve generalization performance. In cross-validation a training set is divided into two or more sets. Training is based on the performance of one set, and generalization performance is estimated by the error on the remaining sets. This technique is computationally intensive in software, but would be easily implemented within the POOKA architecture. As a first step, two fitness metric units could be associated with each node and training data randomly divided. An appropriate tradeoff between errors reported by the two fitness units could then be used to find a potentially more robust solution.

8.2.7 Boosting

Another way to include implicit measures of generalization is suggested by an algorithm known as Boosting [146]. In this scenario, POOKA would be known as a *weak-learner* and would be optimized many times for the same problem. The outputs from each POOKA optimization would then be combined with a weighted sum to produce a final output. This essentially leads to a much more complex classifier. The interesting result from boosting is that this complexity can be increased almost arbitrarily, yet excellent generalization performance is still observed. A key mechanism to the boosting algorithm is an adaptive re-weighting of the training data, which depends on the performance of the previous *weak-leaner*. To incorporate this within the POOKA architecture, a new fitness metric unit would be required that is capable of weighting the error calculated from each pixel.

8.3 Conclusion

Evolutionary Algorithms and Custom Computers have been used independently for a number of years. More recently, these fields have been combined in Evolvable Hardware which means mutual benefit: the long computation times of Evolutionary Algorithms is avoided, and Custom Computer building blocks can be easily optimized. This new design environment seems ideal for producing high performance hardware for solving difficult problems.

This thesis has demonstrated the potential of this design environment for the problem of Automatic Feature Extraction in multi-spectral data sets. In essence, this thesis presents a characterization of the AFE problem in terms of algorithms/theory and Custom Computer hardware resources. This hybrid approach is essential to solving problems of practical interest with Evolvable Hardware.

It should be evident from the large number of potential directions described in Section 8.2 that the future of POOKA is considered bright. This is partly due to the problem addressed. The need for hardware accelerated AFE is felt most in multi-spectral image processing where large volume data sets continue to grow and software poses a computational bottleneck. In other applications, this need is not as great; software speeds are acceptable and algorithm accuracy is the primary requirement.

Another problem, where a POOKA-like approach may be appropriate, is autonomous navigation for real-time robotics. In this case, it is conceivable POOKA could be applied directly at the sensor to find regions of interest in real-time, to be used within a larger robot controller system.

Appendix A

Chapter 6 Developmental Experiments

This section describes some of the more interesting experiments that were performed in the design of the multi-spectral network node of Chapter 6.

A.1 Experiments with Chromosome Representation

The complexity of the node representation leads to the question of how the EA should be best applied. One approach is to directly apply the EA to the chromosome components described in detail in Chapter 6. However, the functionality of the node, particularly the spatial processor, is inspired from image processing algorithms. This suggests an alternative representation that introduces structure to the representation by constraining node configurations to predefined *types*. Table A-3 describes the different types considered for the structured representation. Chromosomes are initialized by randomly selecting both a Spectral and Spatial type. Both components are then configured randomly within a subset of configurations associated with each type. A *typed* Mutation is also implemented, with predefined constraints that maintain the component type during evolution. This is also a probability that a component can change type with mutation during evolution.

Spectral Type	Constraints	Evolved Parameters
Linear Combination	AMM configured as average	Mult_Coefs, Sum_Coefs
Maximum	AMM configured as maximum	Mult_Coefs, Sum_Coefs
Minimum	AMM configured as minimum	Mult_Coefs, Sum_Coefs
Random	No constraints	Entire configuration
Spatial Type		
Average	Rings configured as average	Mult_Coefs, Sum_Coefs
Maximum	Rings configured as average.	Shape of neighborhood through multiplexers
Minimum	Rings configured as average	Shape of neighborhood through multiplexers
Edge	Rings configured as edge magnitude	Mult_Coefs, Sum_Coefs
Texture: Laws	Rings configured as average or edge	Mult_Coefs, Sum_Coefs
Texture: Morphological	Rings configured as maximum or minimum	Mult_Coefs, Sum_Coefs
Random	No Constraints	Entire configuration

Table A-3: Types associated with Node Components

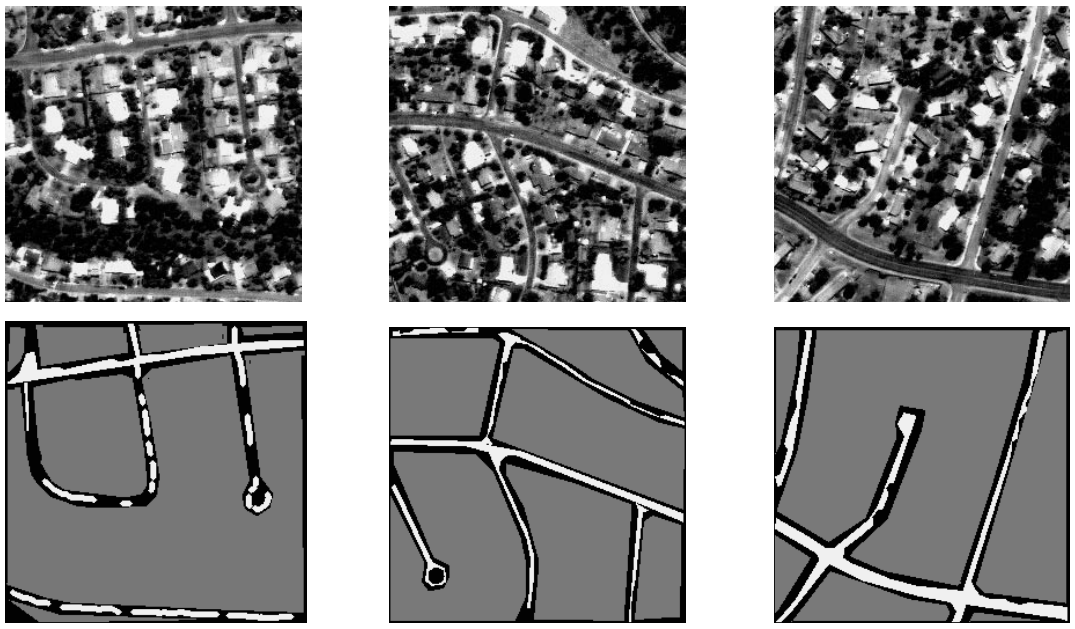


Figure A-1: IKONOS Road Images and Target Classifications: from left Ik1 through Ik3.

Figure A-1 illustrates the problem used to test the EA strategies. These are from the IKONOS instrument, have 4 multi-spectral bands (3 color and 1 near infra-red) and high spatial resolution (1 meter per pixel). The problem is to find the roads in the image. This test problem is of moderate difficulty and therefore a single node could potentially do well. The EA schedule used in these experiments is shown in Table A-4.

EA Parameter	**Number**
Population Size	100
Parents	80
Generations	100
Reproduction	
Elite	10
Mutated Elite	20
Crossover	60
Mutation	90
Random Generation	10

Table A-4: EA Schedule

Random EA	Perfect Classification: 1000											
Training											*Mean*	*S.D.*
ik1	949	960	956	959	952	958	957	953	949	961	955	4.4
ik2	960	964	960	963	966	961	963	970	962	967	964	3.24
Testing												
Ik1-ik2	945	959	960	957	957	956	955	954	958	957	956	4.18
Ik1-ik3	953	919	936	877	901	914	935	921	960	936	925	24.5
Ik2-ik1	934	932	934	921	936	922	938	943	939	936	934	7.03
Ik2-ik3	952	954	958	956	940	949	950	920	946	954	948	11.1
Structured EA												
Training											*Mean*	*S.D.*
ik1	946	940	948	939	938	938	948	951	944	948	944	4.88
ik2	958	965	954	952	961	964	958	955	952	956	958	4.62
Testing												
Ik1-ik2	956	944	948	944	929	945	955	950	951	944	947	7.63
Ik1-ik3	948	942	897	909	928	924	918	964	955	957	934	22.4
Ik2-ik1	901	933	899	888	924	933	923	905	894	893	909	17.2
ik2-ik3	949	958	950	948	959	945	910	950	949	938	946	13.8

Table A-5: Average final fitness scores for 10 runs

Both structured and random GA strategies were each used to evolve node configurations in 10 independent runs. The scores achieved on both training images and test images are summarized in Table A-5.

Discussion

The best 5 runs were taken, and averaged to produce the fitness verse generational plot of Figure A-2. A two-tailed paired sample t-test was also used to investigate the null hypothesis that the two EA strategies, on average, performed identically. The training scores for both images were combined in the t-test. The *p*-value for the null hypothesis on training images was 1.57e-05. This essentially means the results found do not support the null hypothesis i.e. that the means are equal with significant confidence.

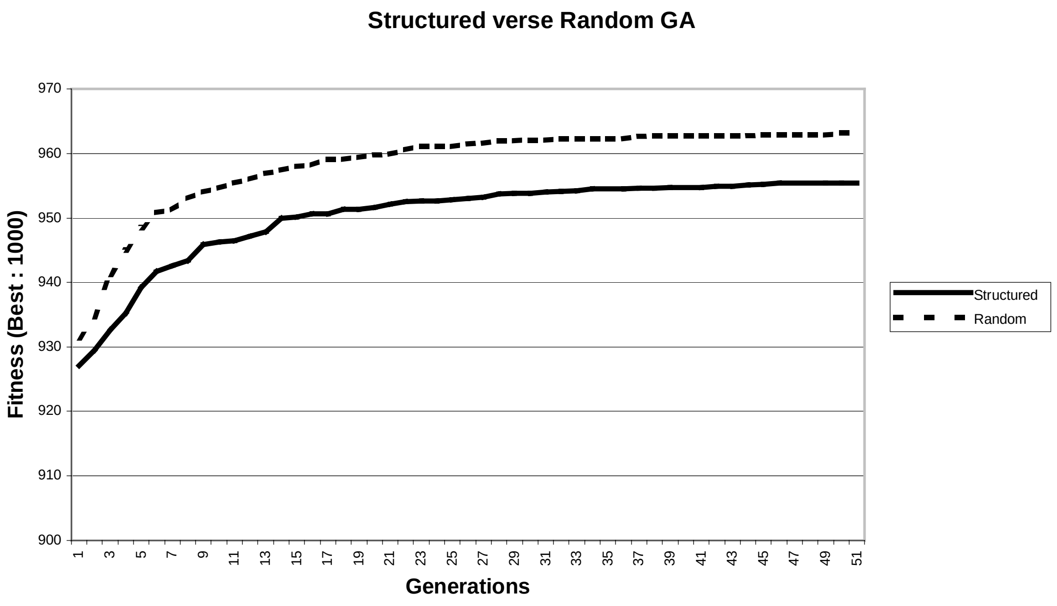


Figure A-2: Fitness verse Generations for Structured and Random Genetic Algorithms

This experiment demonstrated that the EA could be used to evolve the components of the multi-spectral node without using high-level information. It is suggested the node architecture itself constrains the search space enough sufficiently for most problems. The typed approach may be more appropriate if an EA is to be applied to larger chromosomes representing networks of nodes, however since a multiple population based approach was used, this was not considered further.

A.2 Experiments with the Precision Unit

One of the more experimental components of the multi-spectral network node is the Configurable Precision Unit. The motivation for incorporating flexibility within the Precision unit is that limited resources within the FPGA, such as data-path widths, can be optimized by the EA. In this section the idea is explored further by investigating two types of precision units. The first precision unit uses only the tri-state switching logic. In this case, values do not saturate but are simply truncated to a particular 8 bits that the EA selects. This precision unit saves approximately 30 Virtex Slices compared to the one presented in Chapter 6. The second precision unit is the one described in Chapter 6.

A similar experiment to A.2.1 was performed on the IKONOS road finder problems. The two types of Precision unit were implemented. Each node was evolved in 10 separate runs using the evolutionary algorithm parameters described in the previous experiment. The results of the experiment are shown in Table A-6.

Precison Unit with Truncation					Perfect Classification: 1000							
Training											*Mean*	*S.D*
ik1	941	910	894	908	923	893	907	940	955	924	920	21
ik2	946	957	957	956	951	947	946	953	949	938	950	6.1
Testing												
Ik1-ik2	954	919	938	943	940	937	911	948	939	927	936	13
ik1-ik3	937	855	941	950	828	942	950	947	938	931	922	43
ik2-ik1	884	911	899	883	881	878	881	889	874	868	885	12
ik2-ik3	946	946	947	939	924	924	924	937	927	925	934	10
Precision Unit with Saturation												
Training											*Mean*	*S.D.*
ik1	946	940	948	939	938	938	948	951	944	948	944	4.9
ik2	958	965	954	952	961	964	958	955	952	956	958	4.6
Testing												
ik1-ik2	956	944	948	944	929	945	955	950	951	944	947	7.6
ik1-ik3	948	942	897	909	928	924	918	964	955	957	934	22
ik2-ik1	901	933	899	888	924	933	923	905	894	893	909	17
ik2-ik3	949	958	950	948	959	945	910	950	949	938	946	14

Table A-6: Results of Precision Unit Comparison

The output images, from the best solution found, for each training image are illustrated in Figure A-3 and Figure A-4 for the Truncating and Saturating precision units respectively. Note, although the results for each Precision unit look very similar for both training cases, they were found on separate evolutionary runs.

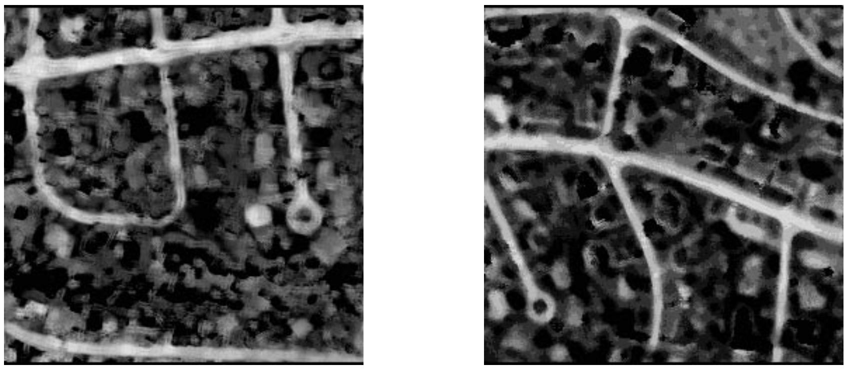


Figure A-3: Best Results on Training Images for truncating Precision Unit (955 and 957)

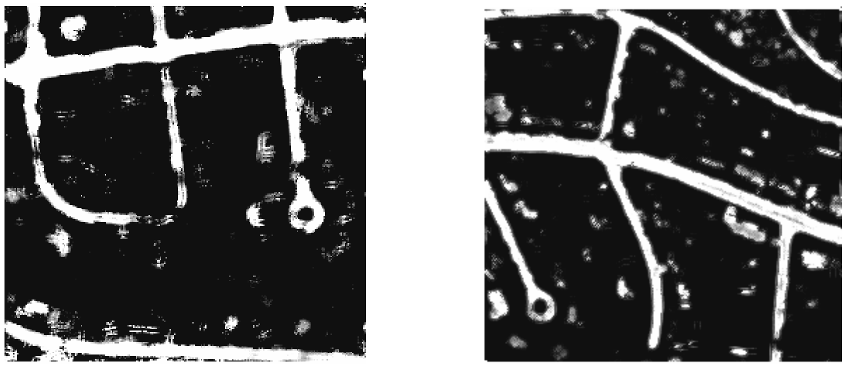


Figure A-4: Best Results on Training Images for Saturating Precision Unit (951 and 965)

Discussion

The results of Table A-6 indicate a slight performance advantage, on this problem, for the node using the Saturating Precision unit. It is interesting to note that both schemes had approximately the same score for their best runs, seen in Figure A-3 and Figure A-4. However, the standard deviation for the Truncating Precision unit was almost twice the variance as the Saturating unit. This suggests that results, comparable in quality to the Saturating Precision unit, could be found using less hardware. However, in this case, it appears such solutions are harder to find using the same evolutionary algorithm.

One reason the saturating precision unit may have performed more consistently is due to the nature of classification problem. The hamming fitness metric effectively thresholds the node output at 0 before the score is calculated. It is likely that thresholding itself is therefore a useful operation within the node. The Saturating Precision unit implements this thresholding much more consistently than the

Truncating Precision unit. In the gray-scale output of Figure A-4 (the final hamming threshold has not been applied) it can be seen that thresholding has been used within the node. This behavior seems well suited to binary classification, and therefore was used in subsequent implementations.

Appendix B

Everything on the Chip

Bibliography

1. Preston, K., *Cellular Logic Computers for Pattern Recognition.* Computer, 1983. **16**(1): p. 36-47.

2. Abramson, D., A.d. Silva, M. Randall, and A. Posutla. *Special Purpose Computer Architectures for High Speed Optimisation*. in *Second Australasian Conference on Parallel and Real Time Systems*. 1995. Fremantle.

3. Miller, J.F., D. Job, and V.K. Vassilev, *Principles in the Evolutionary Design of Digital Circuits - Part I.* Genetic Programming and Evolvable Machines, 2000. **1**(1): p. 7-35.

4. Jain, A.K., *Fundamentals of Digital Image Processing*. Pretice Hall Information and System Sciences Series. 1989, New Jersey: Pretice Hall.

5. Woods, R.C.G.a.R.E., *Digital Image Processing*. 1993, Reading, Massachusetts: Addison-Wesley Publishing Company.

6. Roberts, L.G., *Machine Perception of Three-Dimensional Solids*, in *Optical and Electro-Optical Information Processing*, J.T. Tippet, Editor. 1965, MIT Press: Cambridge, Mass.

7. Sobel, I., *Camera Models and Machine Perception*. 1970, Stanford Artificial Intelligence Lab: Palo Alto.

8. Kirsch, R., *Computer determination of the constituent structure of biological images.* Comput. Biomed. Res., 1971. **4**: p. 315-328.

9. Gabbouj, M., E.J. Coyle, and N.C. Gallagher, *An overview of median and stack filtering.* Circuits, Systems, and Signal Processing, 1992. **11**(1): p. 7-45.

10. Golay, M.J.E., *Hexagonal Parallel Pattern Transformations.* IEEE Trans. Comput., 1969. **C-18**: p. 733-740.

11. Preston, K. and M.J.B. Duff, *Modern Cellular Automata*. 1984, New York: Plenum Publishing Corporation.

12. Serra, J., *Image Analysis and Mathematical Morphology*. 1982, New York: Academic Press.

13. Haralick, R.M., S.R. Sternberg, and X. Zhuang, *Image Analysis Using Mathematical Morphology.* IEEE Trans. Pattern Anal. Machine Intell., 1987. **PAMI-9**(4): p. 532-550.

14. Maragos, P. and R.W. Schafer, *Morphological Filters - Part II: Their Relations to Median, Order-Statistic, and Stack Filters.* IEEE Transactions on Acoustics, Speech and Signal Procesing, 1987. **ASSP-35**(8).

15. Yli-Harja, O., J. Astola, and Y. Neuvo, *Analysis of the Properties of Median and Weighted Median Filters Using Threshold Logic and Stack Filter Representation.* IEEE Transactions on Signal Processing, 1991. **39**(2): p. 395-410.

16. Yu, P. and W. Liao, *Weighted Order Statistics Filters - Their Classification, Some Properties, and Conversion Algorithm.* IEEE Transactions on Signal Processing, 1994. **42**(10): p. 2678-2691.

17. Wendt, P.D., E.J. Coyle, and N.C. Gallagher, *Stack Filters.* IEEE Transactions on Acoustics, Speech and Signal Procesing, 1986. **34**(4): p. 898-911.

18. Bovik, A.C., T. Huang, and D. Munson, *A generalization of median filtering using linear combinations or order statistics.* IEEE Trans. Acoust., Speech, Signal Processing, 1983. **31**: p. 1342-1350.

19. Heinonen, P. and Y. Neuvo, *FIR-median hybrid filters.* IEEE Trans. Acoust., Speech, Signal Processing, 1987. **35**: p. 832-838.

20. Astola, J. and P. Kuosmanen, *Fundamentals of Nonlinear Digital Filtering*. 1997, New York: CRC Press.

21. Schowengerdt, R.A., *Remote Sensing. Models and Methods for Image Processing*. 2 ed. 1997, San Diego: Academic Press.

22. Green, A.A., M. Berman, P. Switzer, and M.D. Craig, *A Transformation for Ordering Multispectral Data in Terms of Image Quality with Implications for Noise Removal.* IEEE Transactions on Geoscience and Remote Sensing, 1988. **26**(1): p. 65-74.

23. Haralick, R.M., *Statistical and Structural Approaches to Texture.* Proceedings of IEEE, 1979. **67**: p. 786-809.

24. Haralick, R. and K. Shanmugam, *Combined spectral and spatial processing of erts imagery data.* Remote Sensing of Environment, 1974. **3**: p. 3-13.

25. Laws, K.I. *Texture energy measures*. in *Proceedings of Image Understanding Workshop*. 1979.

26. Pietikainen, M., A. Rosenfeld, and L.S. Davis, *Experiments with Texture Classification Using Averages of Local Pattern Matches.* IEEE Transactions on Systems, Man and Cybernetics, 1983. **SMC-13**(3).

27. Jain, A.K. and F. Farrokhnia, *Unsupervised Texture Segmentation using Gabor Filters.* Pattern Recognition, 1991. **24**(12): p. 1167-1186.

28. Weska, J.S., C.R. Dyer, and A. Rosenfeld, *A Comparative Study of Texture Measures for Terrain Classification.* IEEE Transactions on Systems, Man and Cybernetics, 1976. **SMC-6**(4).

29. Ojala, T., M. Pietikainen, and D. Harwood, *A Comparative Study of Texture Measures with Classification Based on Feature Distributions.* Pattern Recognition, 1996. **29**(1): p. 51-59.

30. Buf, J.M.H.D., M. Kardan, and M. Spann, *Texture Feature Performance for Image Segmentation.* Pattern Recognition, 1990. **23**(3): p. 291-309.

31. Randen, T. and J.H. Husoy, *Filtering for Texture Classification: A Comparative Study.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 1999. **21**(4): p. 291-309.

32. Tuceryan, M. and A.K. Jain, *Texture Analysis*, in *The Handbook of Pattern Recognition and Computer Vision*, L.F.P. C.H. Chen, P.S.P. Wang, Editor. 1998, World Scientific Publishing Co. p. 207-248.

33. Matheron, G., *Random Sets and Integral Geometry*. 1975, New York: John Wiley and Sons.

34. Maragos, P., *Pattern Spectrum and Multiscale Shape Representation.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 1989. **11**(7): p. 701-716.

35. Dougherty, E.R., *An Introduction to Morphological Image Processing*. Tutorial texts in Optical Engineering, ed. D.C. O'Shea. Vol. TT9. 1992, Washington: SPIE Optical Engineering Press.

36. Dougherty, E.R. and Y. Chen, *Granulometric Filters*, in *Nonlinear Filters for Image Processing*, J.T.A. E.R. Dougherty, Editor. 1999, SPIE Optical Engineering Press / IEEE Press: New York.

37. Gratin, C., J. Vitria, F. Moreso, and D. Seron. *Texture classification using neural networks and local granulometries*. in *2nd International Conference on Mathematical Morphology*. 1994. Fontainebleau: Kluwer Academic Publishers.

38. Aubert, A., D. Jeulin, and R. Hashimoto. *Surface Texture Classification from Morphological Transformations*. in *5th International Symposium on Mathematical Morphology*. 2000. Palo Alto, California: Kluwer Academic Publishers.

39. Duda, R.O., P.E. Hart, and D.G. Stork, *Pattern Classification*. 2 ed. 2001, New York: John Wiley & Sons Inc.

40. Fukunaga, K., *Introduction to Statistical Pattern Recognition*. 1990, San Diego: Academic Press.

41. Bishop, C.M., *Neural Networks for Pattern Recognition*. 1995, Oxford: Oxford University Press.

42. Kruse, F.A., A.B. Lefkoff, J.B. Boardman, K.B. Heidebrecht, A.T. Shapiro, P.J. Barloon, and A.F.H. Goetz, *The Spectral Image Processing System (SIPS) - Interactive Visualization and Analysis of Imaging Spectrometer Data.* Remote Sensing of Environment, 1993. **24**: p. 145-163.

43. Richards, J.A. and X. Jia, *Remote Sensing Digital Image Analysis*. 1999: Springer-Verlag.

44. Rosenblatt, F., *Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*. 1962, Washington D.C.: Spartan.

45. Rumelhart, D.E., G.E. Hinton, and R.J. Williams, *Learning Internal Representations by Error Propagation*, in *Parallel Distributed Processing: Explorations in the Microstructures of Cognition*, D.E.R. et.al, Editor. 1986, MIT press: Cambridge, Mass. p. 318-362.

46. Lippman, R.P., *An introduction to computing with neural nets.* IEEE ASSP Magazine, 1987. **4**: p. 4-22.

47. Fukushima, K., *Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position.* Biol. Cybern, 1980. **36**: p. 193-202.

48. LeCun, Y. and B. Boser, *Convolutional networks for images, speech and time series*, in *The Handbook of Brain Science and Neural Networks*, M. Arbib, Editor. 1995, MIT Press: Cambridge, MA. p. 255-258.

49. Won, Y. and P.D. Gader. *Morphological Shared-Weight Neural Network for Pattern Classification and Automatic Target Detection*. in *IEEE International Conferenec on Neural Networks*. 1995.

50. Tomassini, M., *A Survey of Genetic Algorithms*. Annual Reviews of Computational Physics. Vol. III: World Scientific.

51. Dasgupta, D. and Z. Michalewicz, *Evolutionary Algorithms in Engineering Applications*. 1997, Berlin: Springer-Verlag.

52. Holland, J., *Adaptation in Natural and Artifical Systems*. 1975, Cambridge, MA: MIT Press.

53. Golberg, D.E., B. Korb, and K. Deb, *Messy Genetic Algorithms: Motivation, analysis, and First Results.* Complex Systems, 1989. **3**(5): p. 493-530.

54. Schwefel, H.P., *Evolution and Optimum Seeking*. Sixth-Generation Computer Technology Series. 1995, New York: John Wiley & Sons.

55. Koza, J.R., *Genetic Programming of Computers by Means of Natural Selection*. 1992, Cambridge, Massachusetts: MIT Press.

56. Koza, J.R., *Genetic Programming II. Automatic Discovery of Reusable Programs*. 1994, Cambridge, Massachusetts: MIT Press.

57. Golberg, D., *Genetic Algorithms in Search, Optimization and Machine Learning*. 1989: Addison-Wesley.

58. Banzhaf, W., P. Nordin, R.E. Keller, and F.D. Francone, *Genetic Programming, An Introduction*. 1998, San Francisco: Morgan Kaufmann Publishers Inc.

59. DeJong, K., *Analysis of Behaviour of a Class of Genetic Adaptive Systems*. 1975, University of Michigan: Ann Arbor, MI.

60. Deb, K. and D. Goldberg. *An investigation of niche and species formation in genetic function optimization*. in *Proceedings of the Third International Conference on Genetic Algorithms*. 1989: Morgan Kaufmann.

61. Grosso, P., *Computer Simulations of Genetic Adaptation: Parallel Subcomponent Interaction in a Multilocus Model*. 1985, University of Michigan: Ann Arbor, MI.

62. Kraft, P., N.R. Harvey, and S. Marshall, *Parallel genetic algorithms in the optimization of morphological filters: a general design tool.* Journal of Electronic Imaging, 1997. **6**(4): p. 504-516.

63. Chu, C. *The application of an adaptive plan to the configuration of nonlinear image processing algorithms*. in *Nonlinear Image Processing*. 1990: SPIE.

64. Campbell, N.W. and B.T. Thomas, *Automatic Selection of Gabor Filters for Pixel Classification.* 1995.

65. Saito, H. and M. Mori, *Application of genetic algroithms to stereo matching in images.* Pattern Recognition Letters, 1995. **16**: p. 815-821.

66. Harris, C. and B. Buxton. *Evolving edge detectors with genetic programming*. in *Genetic Programming 1996, Proceedings of the First Annual Conference*. 1996. Cambridge, Massachusetts: MIT Press.

67. Hollingworth, G., A. Tyrrell, and S. Smith. *Simulation of Evolvable Hardware to Solve Low LEvel Image Processing Tasks*. in *Evolutionary Image Analysis, Signal Processing and Telecommunications*. 1999. Goteborg, Sweden: Springer.

68. Ebner, M. and A. Zell. *Evolving a task specific image operator*. in *Evolutionary Image Analysis, Signal Processing and Telecommunications*. 1999. Goteborg, Sweden: Springer.

69. Whitley, D., S. Dominic, R. Das, and C. Anderson, *Genetic Reinforcement Learning for Neurocontrol problems.* Machine Learning, 1993. **13**: p. 259-284.

70. Floreano, D. and F. Mondada, *Automatic creation of an autonomous agent: Genetic evolution of a neural network driven robot.*, in *From Animals to Animats 3: Proc. 3rd Int. Conf. Simulation of Adaptive Behaviour*, D.C. J.A. Meyer, P. Husbands, S. Wilson, Editor. 1994, MIT Press: Cambridge.

71. Harvey, I., P. Husbands, and D. Cliff, *Seeing the light: Artificial evolution, real vision*, in *From Animals to Animats 3: Proc. 3rd Int. Conf. Simulation of Adaptive Behaviour*, J.A.M. D. Cliff, S. Wilson, Editor. 1994, MIT Press.

72. Gurau, F., *Automatic definition of sub-neural networks*. 1994, Labortatoire de I'Informatique du Parallelism, Ecole Normale Superieure de Lyon: Lyon, France.

73. Potter, M.A. and K.A.D. Jong. *Evolving Neural Networks with Collaborative Species*. in *Proceedings of the 1995 Summer Computer Simulation Conference*. 1995. Ontario, Canada.

74. Meeden, L., *An Incremental approach to developing intelligent neural network controllers for robots.* IEEE Transactions on Systems, Man and Cybernetics: Part B, 1996. **26**(3): p. 474-485.

75. Moriarty, D.E. and R. Miikkulaiinen, *Forming Neural Networks through Efficient and Adaptive Coevolution.* Evolutionary Computation, 1998. **5**(4).

76. Yao, X., *A review of evolutionary artificial neural networks.* International Journal of Intelligent Systems, 1993. **8**(4): p. 539-577.

77. Daida, J.M., J.D. Hommes, T.F. Bersano-Begey, S.J. Ross, and J.F. Vesecky, *Algorithm discovery using the genetic programming paradigm: Extracting low-contrast curvilinear features from SAR images of artic ice*, in *Advances in Genetic Programming 2*, K.E.K. P.J. Angeline, Editor. 1996, MIT: Cambridge.

78. Buckles, B.P., F.E. Petry, D. Prabhu, and M. Lybanon, *Mesoscale Feature Labeling from Satellite Images,* in *Genetic Alorithms for Pattern Recognition*, P.P.W. S.K. Pal, Editor. 1996, CRC Press Inc. p. 167-175.

79. Theiler, J., N.R. Harvey, S.P. Brumby, J.J. Szymanski, S. Alferink, S. Perkins, R. Porter, and J.J. Block. *Evolving Retrieval Algorithms with a Genetic Programming Scheme*. in *Proc. SPIE*. 1999.

80. Perkins, S., J. Theiler, S.P. Brumby, N.R. Harvey, and R.B. Porter. *GENIE: A Hyrbid Genetic Algorithm for Feature Classification in Multispectral Images*. in *Proc. SPIE*. 2000.

81. Han, J., C. Moraga, and S. Sinne, *Optimization of feedforward neural networks.* Engineering Applications of Artificial Intelligence, 1996. **9**(2): p. 109-119.

82. Cross, A.D.J., R. Myers, and E.R. Hancock, *Convergence of a hill-climbing genetic algorithm for graph matching.* Pattern Recognition, 2000. **33**(11): p. 1863-80.

83. Xilinx, I., *The Programmable Logic Data Book*. 1994, Xilinx Inc.

84. Altera, *Flex 10KE Product Data Sheet*. 2001.

85. Xilinx, I., *XC6200 Field Programmable Gate Arrays*. 1997, Xilinx Inc.

86. Xilinx, *Virtex Configuration Architecure: Advanced User's Guide*. 1999, Xilinx Inc.

87. Mazor, S., *A Guide to VHDL*. 2nd ed. 1993: Kluwer.

88. Xilinx, *Xilinx Announces Microblaze: World's Fastest FPGA Soft Processor*. 2001.

89. Xilinx, *Xilinx aligns with industry leaders to announce platform FPGA initiative*. 2001.

90. Bertin, P., D. Roncin, and J. Vuillemin, *Programmable Active Memories: a Performance Assessment*. 1993, DIGITAL Paris Research Laboratory.

91. Buell, D.A., J.M. Arnold, and W.J. Kleinfelder, *Spash 2: FPGA's in a Custom Computing Machine*. 1996, California: IEEE Computer Society Press.

92. Corporation, G.O., *SPECTRUM Reconfigurable Computing Platform Documentation*. 1994.

93. Robinson, S.H., M.P. Caffrey, and M.E. Dunham. *Reconfigurable computer array: the bridge between high-speed sensors and low-speed computing.* in *Field-Programmable Logic and Applications. From FPGAs to Computing Paradigm. 8th International Workshop, FPL '98.* 1998. Tallinn, Estonia: Springer-Verlag.

94. Koza, J.R. *Rapid reconfigurable field-programmable gate arrays for accelerating fitness evaluation in genetic programming*. in *Late breaking papers at Genetic Programming 1997 Conference*. 1997.

95. Thompson, A. *Evolving Electronic Robot Controllers that Exploit Hardware Resources*. in *Advances in Artificial Life: Proc 3rd ECAL*. 1995: Springer-Verlag.

96. Kajitani, I., M. Murakawa, D. Nishikawa, H. Yokoi, N. Kajihar, M. Iwata, D. Keymeulen, H. Sakanashi, and T. Higuchi. *An evolvable hardware chip for*

prosthetic hand controller. in *Proc. Seventh Int. Conf. Microelectronics for Neural, Fuzzy and Bio-inspire Systems*. 1999.

97. Murakawa, M., S. Yoshizawa, I. Kajitani, X. Yao, N. Kajihara, M. Iwata, and T. Higuchi, *The GRD Chip: Genetic Reconfiguration of DSPs for Neural Network Processing.* IEEE Transactions on Computers, 1999. **48**(6): p. 628-638.

98. Thompson, A., I. Harvey, and P. Husbands, *Unconstrained Evolution and Hard Consequences*, in *Towards Evolvable Hardware*. 1996, Springer-Verlag. p. 136-165.

99. Zebulum, R.S., A. Stoica, and D. Keymeulen. *The design process of an evolutionary oriented reconfigurable architecture*. in *2000 Congress on Evolutionary Computation*. 2000: IEEE.

100. Haddow, P.C. and G. Tufte. *An evolvable hardware FPGA for adaptive hardware*. in *2000 Congress on Evolutionary Computation*. 2000: IEEE.

101. Stoica, A., D. Keymeulen, R. Tawel, C. Salazar-Lazaro, and W. Li. *Evolutionary Experiments with a Fine-Grained Reconfigurable Architecture for Analog and Digital CMOS Circuits*. in *The First NASA/DoD Workshop on Evolvable Hardware*. 1999. Pasadena, California.

102. Graham, P. and B. Nelson. *A hardware genetic algorithm for the travelling salesman problem on Splash 2*. in *Field-Programmable Logic and Applications*. 1995: Springer: Oxford.

103. Sitkoff, N., M. Wazlowski, A. Smith, and H. Silverman. *Implementing a genetic algorithm on a parallel custom computing machine*. in *Proceedings IEEE Workshop on FPGAs for Custom Computing Machines*. 1995.

104. Kitaura, O., H. Asada, M. Matsuzaki, T. Kawai, H. Ando, and T. Shimada. *A custom computing machine for genetic algorithms without pipeline stalls*. in *IEEE Int. Conf. Systems, Man, and Cybernetics*. 1999.

105. Shackleford, B., G. Snider, R.J. Carter, E. Okushi, M. Yasuda, K. Seo, and H. Yasuura, *A High-Performance, Pipelined, FPGA-Based Genetic Algorithm Machine.* Genetic Programming and Evolvable Machines, 2001. **2**: p. 33-60.

106. Tufte, G. and P.C. Haddow. *Prototyping a GA pipeline for complete hardware evolution*. in *Proc. First NASA/DoD Workshop on Evolvable Hardware*. 1999. Pasadena.

107. Sidhu, R.P.S., A. Mei, and V.K. Prasanna. *Genetic Programming Using Self-Reconfigurable FPGAs*. in *Field Programmable Logic and Applications*. 1999. Glasgow, UK: Springer-Verlag.

108. Yamaguchi, Y., A. Miyashita, T. Maruyama, and T. Hoshino. *A Co-processor System with a Virtex FPGA for Evolutionary Computation*. in *Field Programmable Logic*. 2000. Austria: Springer.

109. Iwata, M., I. Kajitani, H. Yamada, H. Iba, and T. Higuchi. *A Pettern Recognition System Using Evolvable Hardware*. in *Proc. of Parallel Problem Solving from Nature IV - PPSN IV*. 1996. Berlin.

110. Higuchi, T., M. Iwata, D. Keymeulen, H. Sakanashi, M. Murakawa, I. Kajitani, E. Takahashi, K. Toda, M. Salami, N. Kajihara, and N. Otsu, *Real-World Applications of Analog and Digital Evolvable Hardware.* IEEE Transactions on Evolutionary Computation, 1999. **3**(3): p. 220-235.

111. Dumoulin, J., J.A. Foster, J.F. Frenzel, and S. McGrew. *Special Purpose Image Convolution with Evolvable Hardware*. in *Real-World Applications of Evolutionary Computing*. 2000. Edinburgh, UK: Springer.

112. Moon, S. and S. Kong, *Block-Based Neural Networks.* IEEE Transactions on Neural Networks, 2001. **12**(2): p. 307-317.

113. Garis, H.D. and M. Korkin, *The CAM-Brain Machine (CBM): Real Time Evolution and Update of a 75 Million Neuron FPGA-Based Artificial Brain.* Journal of VLSI Signal Processing Systems, 2000. **24**: p. 241-262.

114. Figueiredo, M.A. and C. Gloster. *Implementation of a Probabilistic Neural Network for Multi-spectral Image Classification on an FPGA Based Custom Computing Machine*. in *Vth Brazilian Symposium on Neural Networks*. 1998. Belo Horizonte, Brazil: IEEE Computer Society.

115. Perkins, S., R. Porter, and N. Harvey. *Everything on the Chip: A Hardware-Based Self-Contained Spatially-Structured Genetic Algorithm for Signal Processing*. in *Evolvable Systems: From Biology to Hardware*. 2000. Scotland, UK: Springer-Verlag.

116. Ferguson, L. *Image processing using reconfigurable FPGAs*. in *High-Speed Computing, Digital Signal Processing, and Filtering Using Reconfigurable Logic*. 1996. Boston, USA: SPIE-Int. Soc. Opt. Eng.

117. Rosenfeld, A., *Parallel Image Processing Using Cellular Arrays.* Computer, 1983. **16**: p. 14-20.

118. Kamal, A.K., H. Singh, and D. Agrawal, *A generalized pipeline array.* IEEE Transactions on Computers, 1974. **C-23**: p. 533-536.

119. Sipper, M., *The Emergence of Cellular Computing.* Computer, 1999. **32**(7): p. 18-26.

120. Toffoli, T. and N. Margolus, *Cellular Automata Machines : A new environment for modeling*. 1987, Cambridge, Massachusetts: MIT Press.

121. Wolfram, S., *Theory and Applications of Cellular Automata*, ed. S. Wolfram. 1986, Singapore: World Scientific.

122. Mitchell, M., J.P. Crutchfield, and R. Das. *Evolving Cellular Automata with Genetic Algorithms: A Review of Recent Work*. in *First International Conference on Evolutionary Computation and Its Applications (EvCA'96)*. 1996. Moscow, Russia.

123. Sahota, P., M.F. Daemi, and D.G. Elliman. *Training Genetically Evolving Cellular Automata for Image Processing*. in *International Symposium on Speech, Image Processing and Neural Networks*. 1994. Hong Kong.

124. Sahota, P., M.F. Daemi, and D.G. Elliman. *Using Genetically Evolving Multi-Layer Cellular Automata for Image Processing*. in *Third Golden West Ineternational Conference on Intelligent Systems*. 1995. Netherlands: Kluwer Academis Publishers.

125. Codd, E.F., *Cellular Automata*. 1968, New York: Academic Press.

126. Banks, E.R. *Universality in Cellular Automata*. in *IEEE 11th Annual Symposium on Switching and Automata Theory*. 1970. Santa Monica, California.

127. Chen, K., *Bit-Serial Realizations of a Class of Nonlinear Filters Based on Positive Boolean Functions.* IEEE Transactions on Circuits and Systems, 1989. **36**(6): p. 785-794.

128. Chang, L., W. Fong, and S. Yu. *A New Fast Implementation of Cellular Array for Morphological Filters, Stack Filters and Median Filters ,*. in *Applications of Digital Image Processing XV*. 1992: SPIE.

129. Yu, P. and E.J. Coyle, *The Classification and Assocative Memory Capability of Stack Filters.* IEEE Transactions on Signal Processing, 1992. **40**(10): p. 2483-2497.

130. Lin, L., G.B.A. III, and E.J. Coyle, *Stack filter lattices.* Signal Processing, 1994. **38**: p. 277-297.

131. Yu, P. and E.J. Coyle, *Convergence Behaviour and N-Roots of Stack Filters.* IEEE Transactions on Acoustics, Speech and Signal Procesing, 1990. **38**(9): p. 1529-1544.

132. Paola, J.D. and R.A. Schowengerdt, *A review and analysis of backpropagation neural networks for classification of remotely-sensed multi-spectral imagery.* International Journal of Remote Sensing, 1995. **16**(16): p. 3033-3058.

133. Chung, Y.Y., M.T. Wong, N.W. Bergmann, and M. Bennamoun. *Implementing Neural Network in Custom Computers*. in *IEEE International Conference on Systems, Man and Cybernetics : Conference Theme :*

Intelligent Systems for Humans in a Cyberworld. 1998. San Diego, California, USA: IEEE.

134. Eldredge, J.G. and B.L.Hutchings, *Run-Time Reconfiguration: a method for enhancing the functional density of SRAM-based FPGAs.* Journal of VLSI Signal Processing, 1996. **12**(1): p. 67-86.

135. Sussner, P. *Morphological Perceptron Learning*. in *Joint Conference on the Science and Technology of Intelligent Systems*. 1998. Maryland: IEEE.

136. Ritter, G.X. and P. Sussner. *An introduction to morphological neural networks*. in *13th International Conference on Pattern Recognition*. 1996. Vienna, Austria.

137. Wilson, S.S. *Morphological Networks*. in *Visual Communications and Image Processing IV*. 1989: SPIE.

138. Yang, P. and P. Maragos, *Min-Max Classifiers: Learnability, Design and Application.* Pattern Recognition, 1995. **28**(6): p. 879-899.

139. Radi, A. and R. Poli. *Evolutionary Discovery of Learning Rules for Feedforward Neural Networks with Step Activation Function*. in *Proceedings of the Genetic and Evolutionary Computation Conference*. 1999. Orlando, Florida: Morgan Kaufmann.

140. Lin, J. and E.J. Coyle, *Minimum Mean Absolute Error Estimation over the Class of Generalized Stack Filters.* IEEE Transactions on Acoustics, Speech and Signal Procesing, 1990. **38**(4): p. 663-678.

141. Weber, P.G., B.C. Brock, A.J. Garret, B.W. Smith, C.C. Borel, W.B. Clodius, S.C. Bender, R.R. Kay, and M.L. Decker. *Multispectral Thermal Imager mission overview.* in *Proc. SPIE*. 1999.

142. Harvey, N.R., S.P. Brumby, S. Perkins, J. Theiler, J.J. Szymanski, J.J. Block, R.B. Porter, M. Galassi, and C. Young, *Image Feature Extraction: GENIE vs Conventional Supervised Classification Techniques.* IEEE Transactions on Geoscience and Remote Sensing, 2001.

143. Vapnik, V.N., *Statistical Learning Theory*. Wiley Series on Adaptive and Learning Systems for Signal Processing, Communications, and Control, ed. S. Haykin. 1998, New York: John Wiley & Sons, Inc.

144. S. Perkins, N.R. Harvey, S.P. Brumby, and K. Lacker. *Support Vector Machines for Broad Area Feature Extraction in Remotely Sensed Images.* in *Proc. SPIE 4381*. 2001.

145. Chua, L.O., *CNN: A Paradigm for Complexity*. 1998: World Scientific Publishing Company.

146. Schapire, R.E., Y. Freund, P. Bartlett, and W.S. Lee. *Boosting the margin: a new explanation for the effectiveness of voting methods*. in *Proc. 14th International Conference on Machine Learning*. 1997: Morgan Kaufmann.